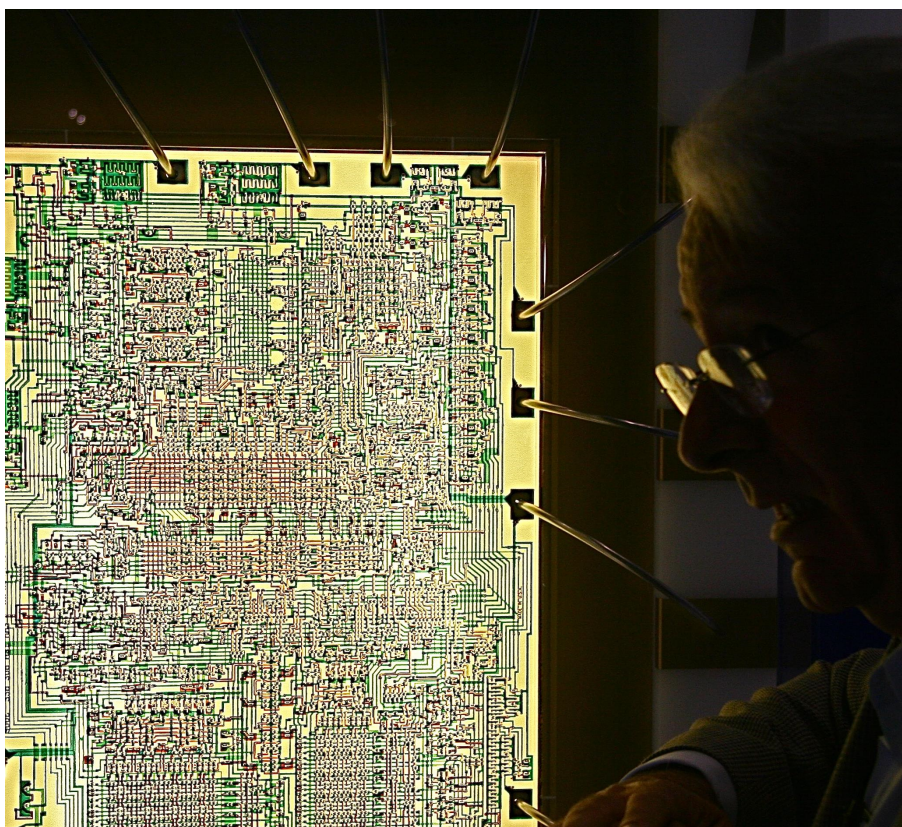
Istituto Statale d'Istruzione Superiore
"Arturo Malignani" - Udine

Istituto Tecnico Industriale "A. Malignani"
Sezione di Telecomunicazione ed Informatica
Dipartimento di Elettronica

Il microcontrollore

Versione 0v10
Settembre 2014
prof. Santino Bandiziol

Indice

Convenzioni adottate nel testo

Il testo normale è scritto utilizzando il presente carattere tipografico e stile di scrittura. Si è cercato di evitare inutili inglesismi ed un eccesso di acronimi, ma trattandosi di appunti tecnici è inevitabile il ricorso all'inglese tecnico e a frequenti abbreviazioni.

Nella prima eventualità, il termine in inglese incontrato per la prima volta viene scritto in corsivo, come nel caso della parola *font* (carattere tipografico). Nell'ipotesi in cui sia ritenuto utile, fra parentesi viene indicata una possibile traduzione che tenga conto del contesto. Le successive volte in cui la stessa parola viene riutilizzata, non verrà più evidenziata in corsivo, dando per acquisito il termine.

Quando lo si ritiene utile la presente convenzione viene usata a "rovescio", ponendo tra parentesi la traduzione inglese di un termine italiano, come nel caso in cui si voglia parlare del carattere tipografico (*font*) e fornirne la traduzione.

L'uso degli acronimi segue regole simili. Viene indicato l'acronimo in grassetto e tra parentesi il suo significato, come nel caso dell'acronimo **PC** (Personal Computer). A volte la parentesi è omessa, come nel caso in cui la spiegazione del significato venga fornita in forma discorsiva e non didascalica. Anche in questo caso, successivi usi dello stesso termine non prevedono nè il grassetto nè l'indicazione del significato posto fra parentesi.

Il corsivo viene utilizzato anche per termini e frasi in italiano che si intendono *enfatizzare*, come nel presente caso. Si è comunque cercato di non abusare di tale convenzione.

L'icona di pericolo è utilizzata per richiamare l'attenzione del lettore su un passaggio particolarmente importante. E' bene, quindi, leggere con attenzione quanto evidenziato dalla presenza del triangolo di pericolo.

Introduzione

I presenti appunti contengono alcune riflessioni sul microcontrollore PIC16F877 e sulla sua programmazione in linguaggio assembly. Essi si prefiggono lo scopo di aiutare l'allievo di un corso di Telecomunicazione e Informatica (con articolazione Telecomunicazioni) nel non facile compito di apprendere un microcontrollore e il relativo linguaggio di programmazione e accrescere la propria autonomia nella risoluzione di problemi di natura logica. Gli appunti non contengono il logico sviluppo verso il linguaggio C, perché tale argomento è trattato in altra documentazione ("Il linguaggio C", dello stesso autore), come pure alcuni prerequisiti (documentazione degli algoritmi, complemento a due e rappresentazione in virgola mobile) che potrebbero tornare utili allo studente.

La scelta di documentare il PIC16F877 è data dal fatto che si tratta di un micro a 8 bit piuttosto semplice, ma che occupa ancora una importante fetta di mercato fra i processori Microchip. Inoltre, una approfondita conoscenza di detto microcontrollore permette una facile migrazione ad altri chip più potenti della stessa casa.

Notazione doverosa. Data la natura della presente documentazione, era inevitabile un pesante e continuo riferimento ai *datasheet* ed alle *application notes* della casa costruttrice. In molti casi le figure (ricopiate con cura, mai "tagliate" e "incollate", in alcuni casi addirittura riviste e corrette) sono riprese pari pari dalla documentazione Microchip, come pure importanti parti della struttura espositiva.

Non si tratta di plagio. L'autore ha voluto fornire uno strumento di studio in lingua italiana del microcontrollore allo studente, senza discostarsi troppo dall'originale, sia per una questione di rispetto verso gli autori che hanno pensato il *layout* del foglio tecnico sia per non disorientare lo studente che vuole cimentarsi con lo studio in lingua inglese.

L'elenco delle figure copiate dalla documentazione Microchip sono elencate in apposita sezione, come di dovere.

Infine un ultimo, un po' nostalgico, appunto. La copertina mostra la maschera dell'Intel 4004, il primo microprocessore della storia. Non è lui, però, il protagonista della copertina. Il vero protagonista è l'uomo che appena si intravvede nell'oscurità: Federico Faggin, il fisico italiano che ha contribuito a cambiare la storia dell'Elettronica. Di lui si è detto e scritto tutto, per cui non si ritiene necessario aggiungere altre parole. Ma c'è un indiretto nesso che collega questo illustre italiano all'ISIS "A. Malignani" di Udine, che forse non tutti conoscono: nei primissimi anni '80, il forse meno famoso fratello minore Giorgio ha avuto modo di insegnare Letteratura Italiana nel nostro Istituto e l'autore

delle presenti pagine ha avuto l'onore di essere suo giovane (allora) collega. E' un collegamento più nostalgico che storico, ma pare giusto salvarlo dall'oblìo.

Le presenti pagine sono state scritte e redatte con cura. Ciò non significa che siano prive di errori o imprecisioni. Quanti volessero segnalare eventuali errori, possono farlo al seguente indirizzo:

bandiziol@katamail.com

Buon lavoro.

Udine, 19/09/2014

prof. Santino Bandiziol

Parte I

L'ARCHITETTURA DEL PIC

Capitolo 1

L'architettura del PIC16F877

Un dispositivo elettronico che occupa un posto di primo piano nel mercato elettronico è la **macchina microprogrammata**. Si tratta di un circuito sequenziale i cui cambiamenti di stato sono dovuti, oltre che dai segnali di ingresso e allo stato precedente, anche da informazioni aggiuntive lette da un dispositivo di memoria. Dette informazioni sono dette **istruzioni**.

La macchina microprogrammata legge dalla memoria di programma dette istruzioni, le decodifica, le interpreta e le esegue. L'esecuzione di tali istruzioni provoca, tenendo conto dello stato precedente e di eventuali segnali di ingresso, i cambiamenti di stato del circuito sequenziale.

A seconda della complessità della macchina microprogrammata e della sua destinazione d'uso se ne possono classificare di diversi tipi. Un primo parziale elenco è il seguente:

- Personal Computer;
- PC industriale;
- PLC;
- sistema a microprocessore;
- sistema a microcontrollore.

Il **Personal Computer** è, insieme, la più versatile e la più diffusa delle macchine microprogrammate. Del suindicato elenco, che non tiene conto di soluzioni sistemiche più avanzate (server, cloud computer, computer quantistici, supercomputer, ecc.), è anche la più potente. Le destinazioni d'uso sono quelle consuete: l'*office automation*, la supervisione d'impianto, la progettazione, ecc.).

Presenta costi piuttosto alti ed offre una notevolissima versatitlità, soprattutto sotto l'aspetto della sua programmazione. Gli ambienti di sviluppo ed i linguaggi possono essere i più disparati ed è possibile scegliere fra una notevole gamma di soluzioni.

Il **PC industriale** (ad es. il PC104) offre la potenza di calcolo e la versatilità del PC, adattata all'ambiente industriale, a costi ragionevoli per la destinazione

d'uso. Si tratta di soluzioni modulari adatte per risolvere problemi di una certa complessità: gestione di reti industriali, controlli di processo complessi, ecc.

Del tutto simili e assimilabili ai PC industriali sono gli *embedded PC*. Si tratta di PC veri e propri il cui hardware è ridotto al minimo indispensabile ed in grado di supportare dei sistemi operativi dedicati e poco voluminosi. Le prestazioni sono ovviamente ridotte, ma i costi sono assolutamente molto contenuti (<100€).

Tale soluzione offre solitamente una maggior robustezza meccanica ed elettrica. E' anch'esso estremamente versatile per quanto riguarda la sua programmazione.

Il **PLC** (*Programmable Logic Controller*) è anch'esso destinato prevalentemente all'ambiente industriale. Rispetto alle due precedenti soluzioni è molto meno versatile, soprattutto dal punto di vista dello sviluppo dei programmi. Solitamente necessità di un proprio linguaggio di programmazione e di un ambiente di sviluppo dedicato. Anche la filosofia di programmazione risulta essere diversa dalle precedenti, il che implica una certa conoscenza del dispositivo.

Presenta costi piuttosto elevati a fronte di caratteristiche meccaniche ed elettriche che si adattno perfettamente all'ambiente industriale. La relativamente scarsa versatilità del PLC è compensata da una vastissima diffusione a livello industriale, che ne fa la macchina ideale in tale contesto.

Il **sistema a microprocessore** segna una differenza concettuale con le macchine microprogrammate viste precedentemente: le precedenti sono da considerarsi dei sistemi complessi con caratteristiche ben definite, mentre quest'ultima è un generico sistema basato principalmente su un semplice componente elettronico. Ciò significa che esso abbisogna di essere affiancato da altri componenti elettronici in modo appropriato, al fine da formare un sistema elettronico completo.

Ciò può sembrare un'inutile complicazione, ma offre un vantaggio inequivocabile: i costi e le funzionalità possono essere dimensionati sulle effettive necessità richieste dal servizio che si vuole offrire. Inoltre, esistono microprocessori che offrono diversi livelli di potenza di calcolo, dal più basso (ed economico) al più alto (e costoso).

Un ulteriore passo verso la semplicità sistemica è rappresentato dal **sistema a microcontrollore**. Il microcontrollore è un componente assolutamente simile al microprocessore, con la fondamentale differenza che incorpora nel proprio *chip* anche la memoria di programma, la memoria dati, solitamente una piccola memoria statica e le *periferiche* di uso più comune.

Per periferiche si intendono quei dispositivi elettronici (configurati come unità di I/O) che coadiuvano l'unità di calcolo in diversi compiti:

- il conteggio di eventi esterni asincroni;
- il conteggio del tempo;
- la conversione analogico/digitale;
- la conversione digitale/analogica;
- la comunicazione seriale;
- la generazione di segnali PWM, ecc.

Siccome non è possibile prevedere un caso standard che comprenda tutte le esigenze del progettista, i microcontrollori sono prodotti con *tagli* di memoria di programma, memoria dati e memoria non volatile diversi, come pure con periferiche differenti.

Il progettista può, in tal modo, scegliere fra una gran varietà di dispositivi con caratteristiche anche assai differenti fra loro e trovare il microcontrollore che meglio soddisfa le esigenze di progetto. I principali vantaggi rispetto al microprocessore sono i costi (molto più bassi a livello di sistema) e lo spazio occupato a parità di memoria e periferiche. L'unico svantaggio, mitigato dalla varietà dei componenti, è dato dall'architettura poco flessibile del microcontrollore.

◊◊◊

Il presente corso analizzerà dettagliatamente un microcontrollore di medio/basse prestazioni e piuttosto diffuso sul mercato: il PIC16F877 prodotto dalla Microchip Technology Inc. Si tratta di un microcontrollore a 8 bit cosidetto *mid range*, ovvero di medie prestazioni in termini di memoria, parallelismo e velocità di elaborazione.

Per uno studio esaustivo del micro si rimanda al relativo *datasheet* in lingua inglese pubblicato dalla Microchip Technology Inc. (cfr. MICROCHIP [5]) da cui è tratta larga parte della seguente documentazione.

1.1 La famiglia PIC16F87x

IL PIC16F877 fa parte, in realtà, di una famiglia di microcontrollori aventi caratteristiche simili, ma "equipaggiati" in maniera leggermente diversa al fine di abbracciare diverse esigenze di mercato.

I micro della famiglia sono 4 e si differenziano, sostanzialmente, per numero di I/O e dimensioni della memoria di programma e della memoria dati, come evidenziato nella tabella 1.1.

Device	Program FLASH	Data Memory	Data EEPROM	I/O pins
PIC16F873	4K	192 bytes	128 bytes	22
PIC16F874	4K	192 bytes	128 bytes	33
PIC16F876	8K	368 bytes	256 bytes	22
PIC16F877	8K	368 bytes	256 bytes	33

Tabella 1.1: Famiglia PIC16F87x

Il presente capitolo tratterà dettagliatamente il PIC16F877, che rappresenta il micro con maggiori linee di I/O e maggior memoria. Presenta, inoltre, una periferica in più rispetto ai PIC16F873 e PIC16F876. La restante parte del

microcontrollore è, però, identica agli altri tre, per cui verrà trattata, in realtà, l'intera famiglia.

In fig. 1.1 è mostrata l'architettura interna del microcontrollore PIC16F87x. La parte evidenziata in neretto è comune a tutti e quattro i micro, mentre la parte colorata in grigio fa parte dell'architettura dei soli PIC 16F874 e PIC16F877.

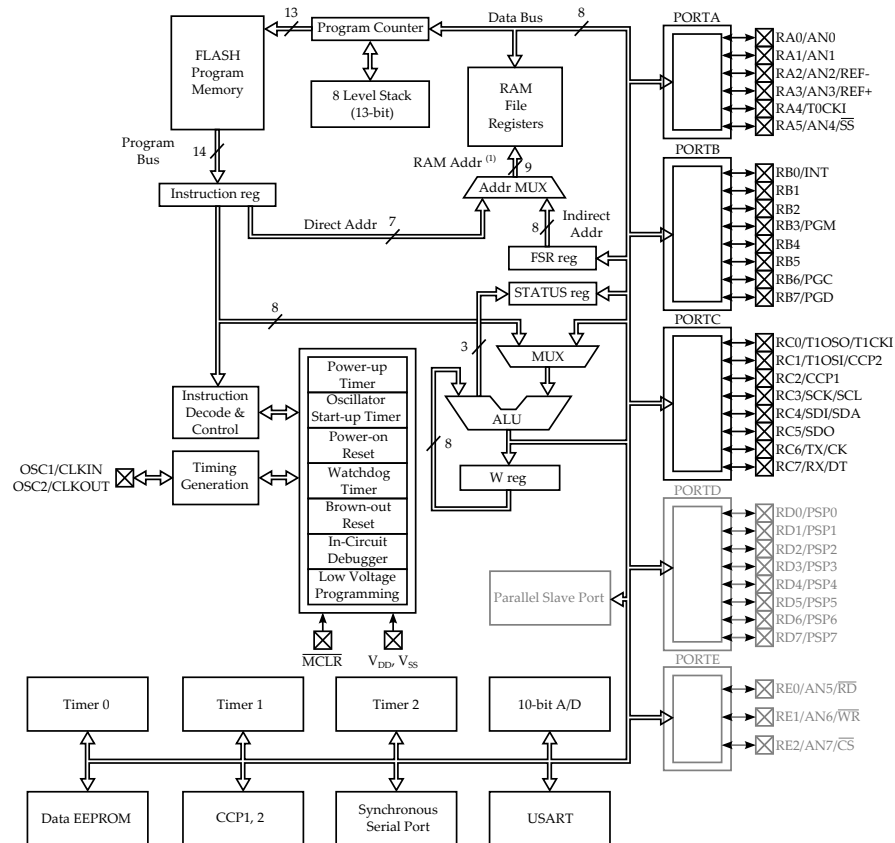



Figura 1.1: Architettura PIC16F87x

Con il termine "architettura" si intende l'insieme dei componenti e delle strutture logiche e fisiche che formano il microcontrollore. Lo studio dell'architettura del micro sarà dettagliata e puntuale, ma per il momento è sufficiente fissare l'attenzione su alcuni aspetti fondamentali che caratterizzano tutti i microcontrollori della famiglia PIC.

Innanzi tutto, i microcontrollori della famiglia PIC sono tutti in tecnologia **RISC** (*Reduced Instruction Set Computer*), in contrapposizione alla precedente tecnologia **CISC** (*Complex Instruction Set Computer*). La scelta di costruire un microcontrollore in tecnologia RISC piuttosto che CISC ha fortissime ripercussioni sull'architettura del dispositivo, per cui è necessario entrare maggiormente nel dettaglio della questione.

1.2 L'architettura Harvard

Fino a qualche decennio fa (inizi anni '80) tutti i microprocessori venivano costruiti in tecnologia CISC. Erano formati da un *set di istruzioni* estremamente esteso (qualche centinaio) che svolgeva dei compiti fortemente dedicati. Ciò significa che le singole istruzioni (ovvero i "comandi" impartiti al microcontrollore che esso è chiamato ad eseguire) non erano di tipo generale ma svolgevano azioni molto specializzate e specifiche.

Ci si è ben presto resi conto che la crescita tecnologica avrebbe comportato un aumento della "disorganizzazione" dei micro, avvicinandolo sempre più ad una "somma di specificità" piuttosto che ad un dispositivo flessibile ed ugualmente capace di eseguire velocemente sia istruzioni semplici che complesse, sia dedicate che di tipo generale.

A queste richieste di flessibilità e velocità si è cercato di rispondere con la tecnologia RISC. Essa presenta, grossolanamente, le seguenti caratteristiche:

- architettura semplice e lineare;
- set di istruzioni ridotto;
- codifica delle istruzioni di lunghezza costante;
- metodi di indirizzamento semplici;
- tempi di accesso alle memorie ridotto;
- *pipelining* delle istruzioni. [1]

Esse sono ottenute implementando nel *chip* un'architettura di tipo **Harvard**, che si contrappone tecnologicamente alla vecchia architettura **Von Neumann**.

Quest'ultima è caratterizzata, fra l'altro, da essere internamente interconnessa da tre *bus* distinti:

- bus degli indirizzi;
- bus dei dati;
- bus di controllo.

Tale soluzione, però, presenta un inconveniente: il bus dei dati è preposto al trasporto sia dei dati da e verso l'unità di calcolo che delle istruzioni verso l'unità preposta alla loro decodifica.

Ciò implica che la lettura e l'esecuzione delle istruzioni avvenga serialmente nel seguente ordine:

- lettura dell'istruzione attraverso il bus dei dati;
- decodifica dell'istruzione;
- eventuale lettura del dato dalla memoria attraverso il bus dei dati;
- eventuale modifica del dato;
- eventuale scrittura del dato dalla memoria attraverso il bus dei dati.

Una siffatta sequenza di azioni rende l'esecuzione delle istruzioni da parte del micro estremamente lenta dato che sia le istruzioni che i dati sono veicolati all'unità di calcolo attraverso lo stesso bus.

L'architettura Harvard, invece, rende l'esecuzione delle istruzioni più veloce grazie alla presenza di due distinti bus: uno per i dati (*Data Bus*) ed uno per

[1] Sul concetto di *pipelining* e di indirizzamento si tornerà dettagliatamente nei prossimi capitoli.

le istruzioni (*Program Bus*). In tal modo, dopo aver letto l'istruzione, è possibile eseguirla (accedendo se necessario alla memoria mediante il bus dei dati) mentre contemporaneamente si legge l'istruzione successiva (mediante il bus delle istruzioni).

La separazione fra detti bus è evidenziata nella fig. 1.2.

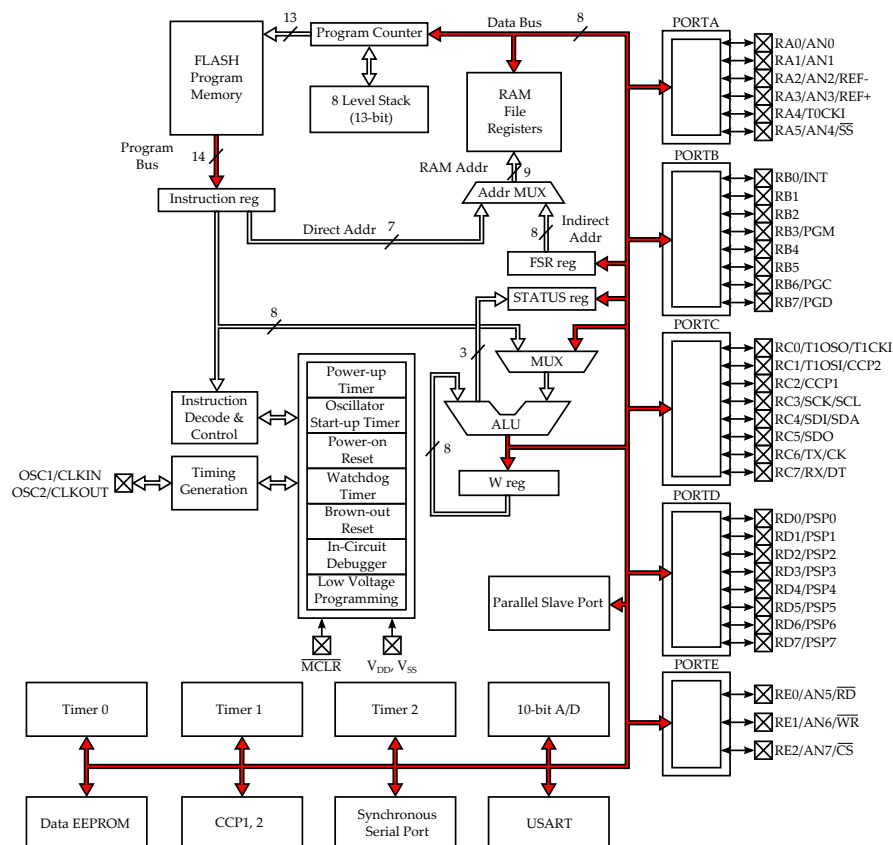



Figura 1.2: Data Bus e Program Bus

Si noti come il Program Bus sia formato da 14 bit, in modo da poter contenere in una sola parola sia il codice operativo dell'istruzione che l'eventuale operando.

1.3 L'ALU

L'unità di calcolo del microcontrollore è rappresentato dall'ALU (*Aritmetic Logic Unit*). Detta unità (evidenziata in rosso in fig. 1.3) ha il compito principale di eseguire tutte le operazioni aritmetico-logiche e di rendere il risultato di dette operazioni disponibili sul bus dei dati e per il registro di lavoro W (*working register*).

Si noti il percorso attraverso il quale l'ALU acquisisce i due operandi dell'operazione aritmetico-logica: uno dei due operandi è sempre fornito dal registro W (evidenziato in rosa), mentre il secondo operando provviene, attraverso l'apposito multiplexer (evidenziato in rosa), o dal bus dei dati (ovvero da una periferica o dalla memoria) oppure direttamente dall'*instruction register*.

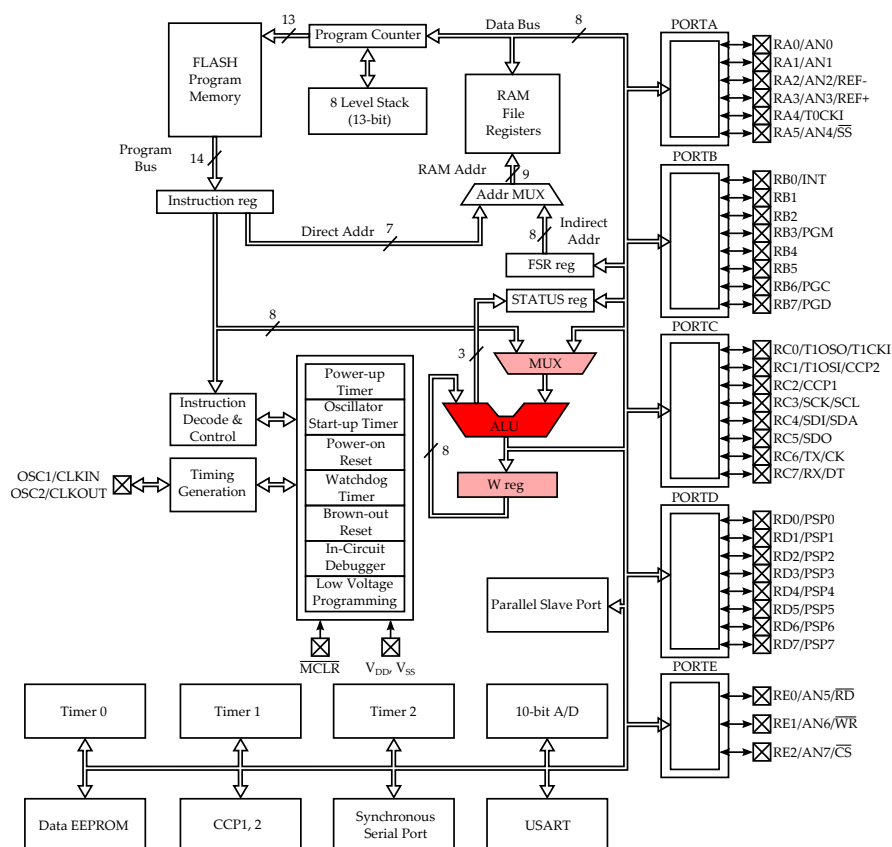



Figura 1.3: L'ALU

Ciò è dovuto al fatto tutte le operazioni aritmetico-logiche prevedono l'uso del registro di working. In altre parole, uno dei due operandi deve essere contenuto nel registro W.

L'altro operando ha una provenienza più articolata. Se esso è una costante può essere fornito direttamente attraverso l'istruzione. Potrebbe, però, essere una variabile, nel qual caso il secondo operando potrebbe essere fornito da una qualsiasi periferica (il convertitore A/D, un Port, un Timer, ecc.) oppure dalla memoria dati (*RAM File Registers*).

Si noti, infine, la destinazione del risultato elaborato dall'ALU: esso può essere rappresentato o dal registro W, da una delle periferiche o dalla memoria dati.

1.4 Il *Program Counter*

Un componente fondamentale di ciascun micro è il *program counter* o contatore di programma. Per comprendere la funzione di tale componente si deve pensare a come funziona un macchina microprogrammata.

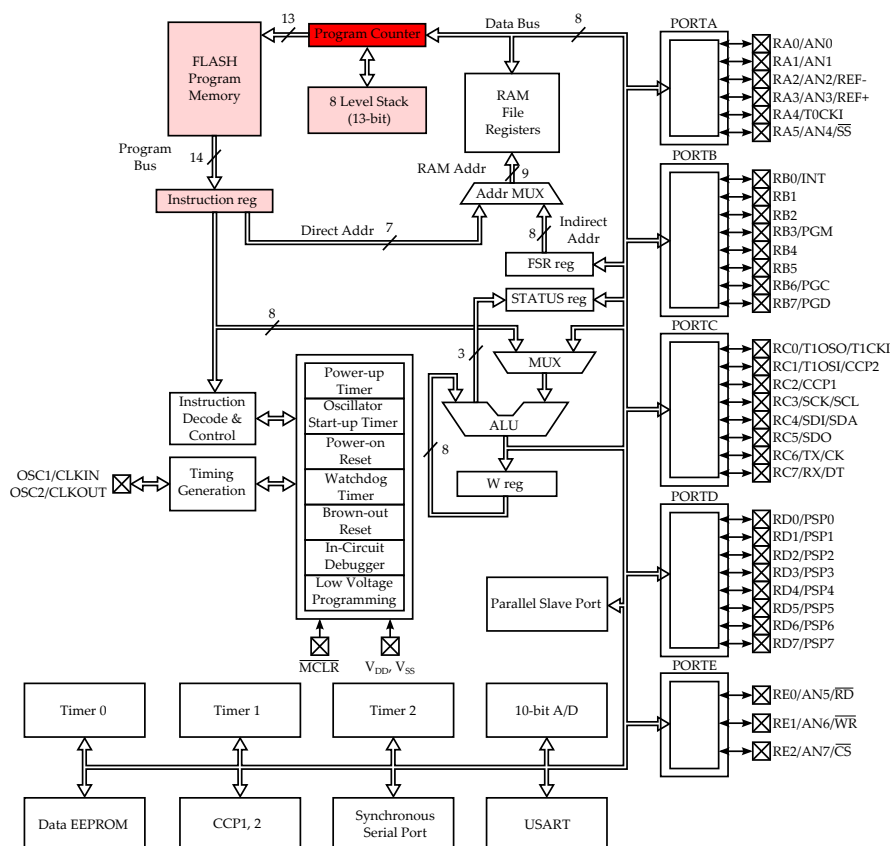



Figura 1.4: Il Program Counter

Il program counter ha il compito di selezionare l'esatto indirizzo di memoria di programma da cui prelevare l'istruzione da eseguire. Tale indirizzo è determinato da due diverse entità: l'ALU e lo *Stack* (vedi fig. 1.4). Il primo ha il compito di aggiornare il program counter ogni qualvolta è necessario calcolare aritmeticamente uno **spiazzamento**. Lo Stack, invece, ha il compito di fornire al PC **l'indirizzo di ritorno** alla fine dell'esecuzione di una subroutine. In tutti gli altri casi il Propram Counter è in grado di calcolare autonomamente il nuovo indirizzo, o perché l'istruzione corrente è un'istruzione di salto (quindi il PC calcola da solo lo spiazzamento) o semplicemente perché va indirizzata l'istruzione che segue quella in esecuzione.

A tal proposito si richiama l'attenzione sulla doppia freccia che unisce lo Stack al PC. Il simbolo indica che vi è scambio di informazione fra i due registri in entrambi i sensi. Infatti, quando il Program Counter indirizza un'istruzione di salto a subroutine **comunica** allo Stack l'indirizzo di ritorno. Quando, in-

vece, il PC indirizza un'istruzione di ritorno da subroutine **chiede** allo Stack l'indirizzo di ritorno.

Si noti che il program counter indirizza la memoria di programma con 13 linee. Ciò significa che sono indirizzabili $2^{13} = 8k$ istruzioni. Tale indicazione è evidentemente valida solo per il PIC16F876 e il PIC16F877. Il program counter dei restanti due micro sarà effettivamente composto da sole 12 linee.

Una volta indirizzata, la FLASH Program Memory veicola verso l'Instruction Register, attraverso il bus di programma, la singola istruzione da 14 bit.

Questo dettaglio introduce il prossimo argomento.

1.5 Il *pipelining*

Le istruzioni fornite dalla FLASH Program Memory all'Instruction Register sono tutte di 14 bit e sono conformi alla filosofia RISC: sono cioè semplici e di tipo generale ossia non eccessivamente specialistiche. Tali caratteristiche permettono al micro di eseguire ciascuna istruzione in 4 cicli di clock, ovvero 1 ciclo macchina.

La separazione del bus di programma con quello dei dati permette, inoltre, il *pipelining*, come evidenziato in fig. 1.5.

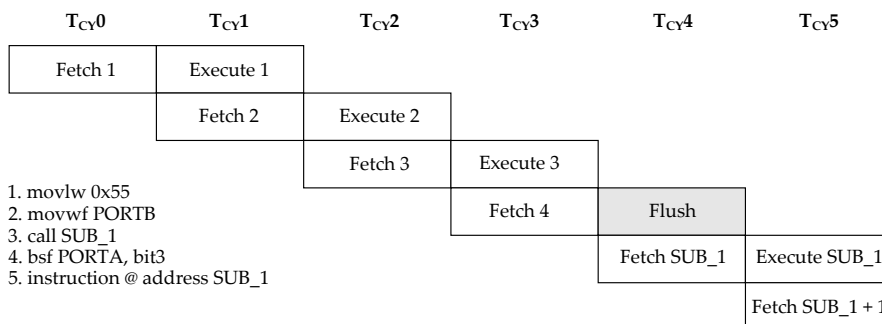



Figura 1.5: Pipelining

Esso funziona nel seguente modo.

Si suppongano le 5 istruzioni indicate[2] in figura e la relativa rappresentazione grafica. Durante il ciclo macchina $T_{CY}0$ viene eseguito il *fetch* (ovvero la lettura) dell'istruzione 1. Detta istruzione viene decodificata ed eseguita durante il ciclo macchina successivo $T_{CY}1$. Contemporaneamente all'esecuzione dell'istruzione 1 viene letta l'istruzione 2. Tale azione si rende possibile a causa della separazione fra i due suddetti bus.

L'esecuzione dell'istruzione 1, infatti, potrebbe implicare un accesso, attraverso al bus dei dati, ad esempio, alla memoria dati, e se lo stesso bus dovesse

[2]Non è necessario comprendere il significato delle singole istruzioni visualizzate in fig. 1.5. E' sufficiente tener presente che l'istruzione 3 rappresenta una chiamata a subroutine.

essere usato per eseguire il fetch dell'istruzione 2, quest'ultima azione dovrebbe attendere il termine dell'esecuzione dell'istruzione 1, allungando i tempi di elaborazione del micro.

Analogamente, durante il ciclo macchina $T_{CY}2$, viene eseguita l'istruzione 2 e caricata l'istruzione 3. Quest'ultima, però, implica uno spiazzamento, ovvero un salto ad un'altra locazione di memoria, potenzialmente anche molto distante dalla precedente. Una simile azione non può quindi essere eseguita in un solo ciclo macchina: infatti, le istruzioni che implicano un salto sono eseguite in due cicli macchina anzichè uno solo.

Infatti, la decodifica dell'istruzione 3 avviene durante il caricamento dell'istruzione 4 (ciclo macchina $T_{CY}3$), quindi quando il micro si renderà conto che l'istruzione 3 implica un salto ad istruzioni situate in tutt'altra parte della memoria di programma, sarà già troppo tardi.

Il micro rimedierà eseguendo un *flush* (rimozione) dell'istruzione 4 evitando di eseguirla e caricherà (ciclo macchina $T_{CY}4$) la prima istruzione della subroutine. Questa verrà eseguita durante il ciclo macchina $T_{CY}5$, insieme al caricamento della seconda istruzione della subroutine.

Il meccanismo di pipelining ed il conseguente *flushing* diventa un po' più subdolo in presenza di alcune istruzioni particolari. Si supponga le istruzioni del listato 1.1:

Listing 1.1: Ciclo in linguaggio assembler

```
        movlw 0x03
        movwf Counter
Ciclo:  decfsz Counter
        goto Ciclo

        movlw 0x05
        ;ecc. ecc.
```

L'istruzione `decfsz Counter` (*decrement file register and skip if zero*) decrementa la variabile Counter e se, dopo il decremento, la variabile è uguale a zero, salta (*skip*) l'istruzione successiva (`goto Ciclo`), altrimenti no. Quindi l'istruzione `decfsz Counter` per due volte esegue l'istruzione che la segue e la terza volta esegue il salto, uscendo in tal modo dal ciclo per proseguire sequenzialmente con il programma.

La stessa istruzione si comporta in modi diversi: quando esegue il salto il pipelining necessita di un flush, altrimenti no. Di ciò si deve tener conto se il ciclo, ad esempio, è utilizzato per creare un loop di ritardo, dato che in presenza di flushing l'istruzione non viene più eseguita in un ciclo macchina, ma in due.

Il meccanismo di pipelining permette il raggiungimento velocità di elaborazione più alte ed aumenta, nel complesso l'efficienza della macchina.

1.6 I modi di indirizzamento

Si è visto nei paragrafi precedenti che una delle peculiarità dei sistemi RISC è la semplicità di indirizzamento. Sostanzialmente i micro della famiglia PIC16F87x permettono tre diversi tipi di indirizzamento:

- indirizzamento diretto;
- indirizzamento indiretto;
- indirizzamento relativo.

Si ha indirizzamento diretto quando la variabile è indirizzata esplicitamente mediante l'istruzione (ad es. `movwf Counter`, ossia "sposta il contenuto del registro di working nel file register Counter"). L'indirizzamento relativo è utile per la lettura in tabella mentre l'indirizzamento indiretto richiede l'uso del registro FSR.

1.6.1 Indirizzamento diretto

Si faccia riferimento alle seguenti istruzioni del listato 1.2:

Listing 1.2: Esempio di indirizzamento diretto

```
Memory  equ     0x20    ;Locazione di memoria
Value   equ     0x55    ;Costante
...
        movlw   Value
        movwf   Memory
        ;ecc. ecc.
```

Le prime due righe sono due direttive e servono a stabilire un'equivalenza fra etichette e valori. Quindi `Memory` è un'etichetta che assume il valore costante $0x20$, mentre `Value` è una costante che assume il valore costante $0x55$.

La traduzione in linguaggio corrente della prima istruzione è la seguente:

"Carica nel registro di lavoro W la costante Value"

Quindi nel registro di lavoro viene caricata la costante $0x55$. La traduzione in linguaggio corrente della seconda istruzione suona come segue:

"Sposta il contenuto del registro W nella locazione puntata da Memory"

Quindi la locazione di memoria di indirizzo $0x20$ verrà riempita con il valore $0x55$. Si noti, però, che `Memory` e `Value` sono solo due etichette che rappresentano altrettante costanti. Il significato che esse assumono è dato dall'istruzione e non dalla direttiva in sé. Si veda l'esempio seguente:

Listing 1.3: Esempio di lettura-scrittura

```
        movf    Memory, W
        movwf   Value
        ;ecc. ecc.
```

La prima istruzione non carica la costante `Memory` in W, ma il **contenuto** della locazione puntata da `Memory`. Detto valore viene poi copiato nella locazione puntata da `Value`.

1.6.2 Indirizzamento indiretto

L'indirizzamento indiretto utilizza il registro FSR (vedi fig. 1.6).

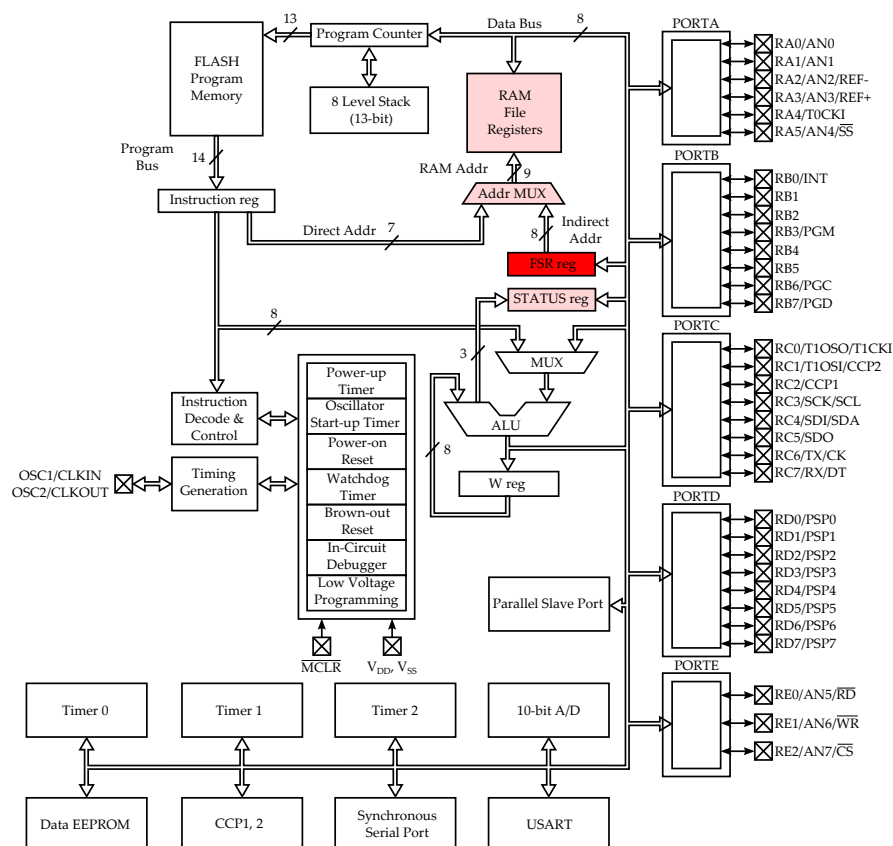



Figura 1.6: Il registro FSR

Si ha indirizzamento indiretto quando l'accesso alla locazione da leggere o scrivere non avviene direttamente attraverso l'istruzione, ma attraverso i registri FSR e INDF. Il registro FSR funge da puntatore alla locazione da scrivere (tipicamente una locazione di memoria dati) e può essere decrementato o incrementato, modificando in tal modo l'indirizzo puntato.

Il registro INDF, in realtà, non è un registro fisico ma semplicemente un'espressione sintattica utile all'istruzione per indicare un indirizzamento indiretto.

Concorre alla formazione dell'indirizzo anche il registro di stato (*Status Register*). Esso infatti fornisce, con un meccanismo che verrà illustrato il prossimo paragrafo, il nono bit dell'indirizzo di memoria (vedi fig. 1.6)

Il modo più agevole per illustrare l'indirizzamento indiretto è attraverso un esempio. Anche in questo caso sarà necessario presentare all'allievo delle istruzioni, che ancora non conosce. Non è importante che siano comprese in quanto tali. E' sufficiente comprendere come è usato il registro FSR nell'indirizzamento della memoria e quale è il ruolo del regitro INDF. L'esempio è tratto direttamente dal datasheet Microchip (cfr. MICROCHIP [5]).

Si tratta di alcune istruzioni che permettono l'azzeramento delle locazioni di memoria comprese fra $0x20$ e $0x2F$.

Listing 1.4: Azzeramento di 16 locazioni di memoria

```
1.               movlw   0x20    ;Inizializza il puntatore FSR
2.               movwf   FSR
3.       Ciclo:  clrf    INDF    ;Azzera locazione puntata da FSR
4.               incf    FSR     ;Aggiorna il puntatore
5.               btfss   FSR, 4  ;Fine azzeramento?
6.               goto    Ciclo
7.               ;ecc. ecc.
```

L'istruzione 1. permette il caricamento del valore $0x20$ (ossia 32 in decimale) nel registro di lavoro W, mentre mediante l'istruzione 2. si trasferisce detto valore nel registro FSR. Il caricamento diretto di FSR non esiste (le istruzioni di tipo RISC sono di tipo generale, non specialistico) per cui si deve passare attraverso il registro di lavoro.

Dopo le prime due istruzioni, quindi, il registro FSR punta la locazione $0x20$ nella memoria dati (*RAM File Registers*).

La terza istruzione azzera (`clrf`: *clear file register*) la locazione di memoria **puntata da FSR**.

Lo studente potrebbe legittimamente chiedersi perché non si è direttamente scritto `clrf FSR`: perché una siffatta scrittura avrebbe utilizzato l'indirizzamento diretto, azzerando non la locazione puntata da FSR, ma direttamente il registro FSR! Per distinguere i due indirizzamenti si è ricorsi, quindi, ad un'espressione sintattica piuttosto che ad un registro fisico vero e proprio.

Si invita l'allievo a soffermarsi su tale punto e di comprendere bene il funzionamento dell'istruzione 3. prima di proseguire.

L'istruzione 4. incrementa il registro FSR in modo che esso punti la successiva locazione di memoria (nel caso presente $0x21$), mentre l'istruzione 5. testa se FSR ha raggiunto il valore $0x30$ (`btfss`: *bit test file and skip if set*, ossia testa il bit - indicato in argomento dell'istruzione - del registro e salta se è a 1). Si noti che in binario il valore esadecimale $0x30$ equivale a 00110000, mentre $0x2F$ equivale a 00101111. Testando il bit 4 del registro FSR si è in grado di stabilire se vi è stato passaggio fra $0x2F$ e $0x30$.

Se il bit 4 di FSR non è posto a 1 (che indica la fine del ciclo), l'istruzione 6. non viene saltata a si torna all'istruzione 3, finché tutte e 16 le locazioni di memoria non sono state azzerate.

In tal modo, con poche istruzioni, si è potuto operare su una vasta zona di memoria in maniera semplice ed efficiente.

1.6.3 Indirizzamento relativo

L'indirizzamento relativo verrà trattato diffusamente quando si parlerà di lettura in tabella. Nel presente paragrafo l'argomento verrà trattato in maniera non approfondita, mancando ancora molte nozioni utili in termini di linguaggio Assembly.

L'indirizzamento relativo consiste nel creare artificialmente uno spiazzamento, solitamente **sommando al contatore di programma un determinato valore**.

Un esempio esplicativo può essere formato dalle seguenti istruzioni:

Listing 1.5: Modifica del *Program Counter*

```
1.              movlw   0x06    ;Carica W con l'offset (pari)
2.              addwf   PCL     ;Somma l'offset al PC
3.              movlw   0x00    ;Tabella di W al quadrato
4.              goto    Fine    ;Esce dalla tabella
5.              movlw   0x01
6.              goto    Fine
7.              movlw   0x04
8.              goto    Fine
9.              movlw   0x09
10.             goto    Fine
11.             ;ecc. ecc.
xx. Fine:       ;Continua il programma
```

Le suddette istruzioni permettono di calcolare (utilizzando una rudimentale tabella) il quadrato del valore di W diviso per 2. Il risultato viene messo in W. In simboli ciò equivale a:

$$W \Leftarrow \left(\frac{W}{2}\right)^2 \tag{1.1}$$

L'istruzione 1. carica nel registro W il valore (moltiplicato per 2) di cui si vuole calcolare il quadrato (quindi nel nostro caso particolare il valore 3). L'istruzione 2. rappresenta il cuore dell'algoritmo: al valore corrente del contatore di programma (che contiene l'indirizzo dell'istruzione seguente, ovvero la 3.) viene sommato il valore di W ed il risultato viene posto nel contatore di programma (ovvero in PCL [3]).

In tal modo, dopo la somma, il PCL conterrà l'indirizzo (se $W = 0x06$) dell'istruzione 9. e non di quella che sarebbe stata eseguita in assenza della suddetta somma.

La tecnica appena illustrata permette di implementare degli algoritmi di calcolo molto efficienti, dei quali si avrà modo di discutere diffusamente nelle prossime pagine. Per ora è stato sufficiente illustrare molto brevemente la terza modalità di indirizzamento del micro.

[3] In realtà il registro PCL rappresenta la parte bassa del contatore di programma, ovvero gli 8 bit meno significativi. Si tornerà esaustivamente su tale dettaglio nei prossimi paragrafi.

1.7 La memoria di programma

Il presente paragrafo tratterà un aspetto particolare della memoria di programma: la sua suddivisione in **pagine**. In fig. 1.7 sono evidenziati i tre blocchi che concorrono alla determinazione dell'indirizzo finale di memoria: il PC, lo Stack e lo Status. Il quarto blocco è quello principale, ovvero la memoria vera e propria.

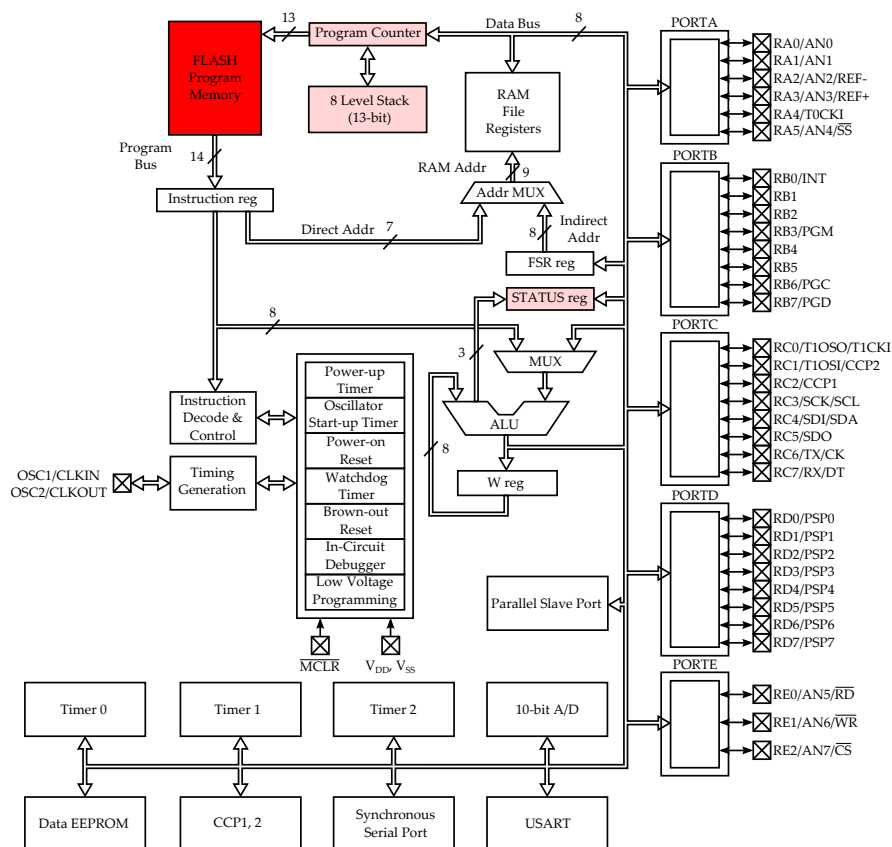



Figura 1.7: La memoria di programma

Per motivi tecnologici e di costo la memoria di programma è divisa in 2 parti (PIC16F873 e PIC16F874) oppure in 4 parti (PIC16F876 e PIC16F877). Tale suddivisione crea molti problemi al tecnico che deve scrivere il software per cui va compresa molto bene.

La memoria è indirizzata dal contatore di programma mediante 13 bit, ma solamente 11 bit sono forniti al PC attraverso le istruzioni. Mediante detti 11 bit si possono indirizzare una qualsiasi fra 2048 locazioni di memoria, che tradotto in esadecimale equivale a 0x0800.

Ciò significa che con gli 11 bit forniti al PC attraverso le istruzioni di `goto` oppure di `call` si può indirizzare una qualsiasi locazione posta all'interno

della singola pagina. La selezione della singola pagina avviene mediante 2 bit [4] posti nel registro di stato del microcontrollore.

Una mappatura esplicativa della memoria è evidenziata in fig. 1.8.

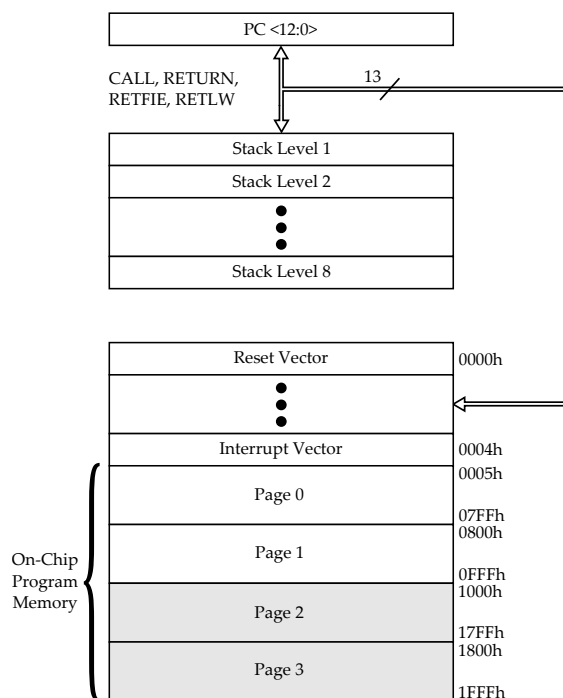



Figura 1.8: Mappatura della memoria di programma

In detta figura appaiono evidenziate due aree fondamentali della memoria: lo stack e le prime 5 locazioni della pagina 0 della memoria di programma. Di esse si parlerà brevemente nei prossimi due paragrafi.

1.7.1 Lo *Stack*

Letteralmente *Stack* significa pila o catasta. Si tratta di un’area di memoria utilizzata dal PIC durante l’esecuzione delle subroutine.

Quando un microcontrollore deve eseguire un salto a subroutine deve in qualche maniera “ricordarsi” qual è l’indirizzo di ritorno al quale saltare quando la subroutine finisce. Non è, infatti, sufficiente porre alla fine della routine l’istruzione `goto`, dato che la stessa subroutine potrebbe essere chiamata da diversi indirizzi.

Non solo. Le routine potrebbero essere chiamate anche in maniera **asincrona**, mediante una procedura di **interruzione** (*interrupt*), che ha la particolarità di interrompere il programma in un punto qualsiasi non predeterminabile.

[4] 1 solo bit nel caso dei PIC16F873 e PIC16F874, che possiedono solamente 2 pagine da 2048 byte.

Al termine della routine di interruzione sarebbe, quindi, impossibile determinare l'indirizzo di ritorno al quale saltare se non si fosse memorizzato detto indirizzo in un'appostita area di memoria: lo Stack.

In fig. 1.9 è evidenziato graficamente l'evolversi dell'uso dello Stack durante l'esecuzione delle istruzioni.

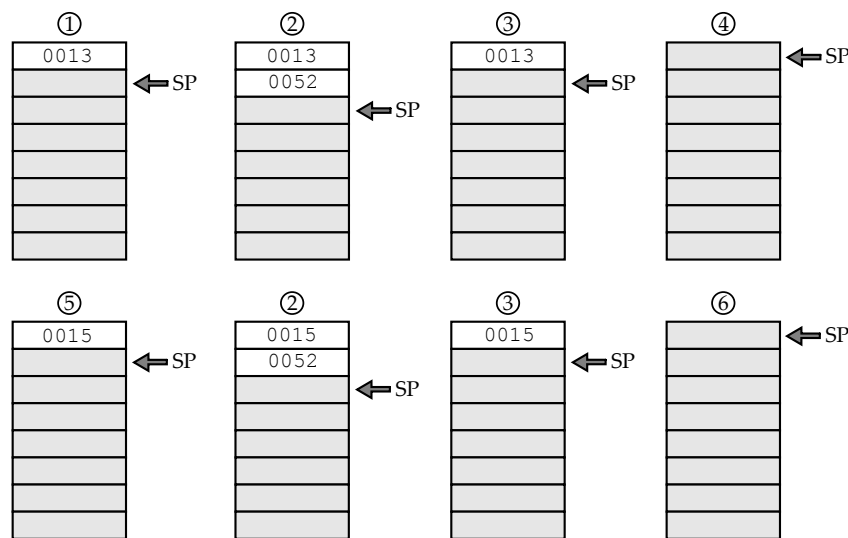



Figura 1.9: Il funzionamento dello Stack

Le prime due istruzioni provvedono ad azzerare la variabile *Pippo* e a carica il valore 0*x*03 nel registro di lavoro. Si suppone, a tal punto dell'esecuzione del programma, che lo Stack non sia utilizzato. La terza istruzione richiama la subroutine *SUB*_1. Ciò implica il salvataggio dell'indirizzo di ritorno (1), che avviene memorizzando il valore 0013 nel livello 1 dello Stack e spostando lo *Stack Pointer* nel livello 2 dello Stack [5].

Dopo aver memorizzato l'indirizzo di ritorno, il micro esegue il salto alla subroutine *SUB*_1, esegue la prima istruzione (`addwf Pippo`) e poi prepara il salto alla subroutine *SUB*_2, memorizzando nel livello 2 dello Stack l'indirizzo di ritorno dalla *SUB*_2 (2) e spostando lo SP nel livello 3.

Dopo che il micro ha eseguito la prima istruzione della *SUB*_2 (`incf Pippo`) deve prepararsi ad eseguire l'istruzione `return` che permette di saltare all'istruzione seguente la `call` che ha permesso il salto alla subroutine. Ciò avvie-

[5]Si noti che lo *Stack Pointer* non compare nell'architettura del PIC16F87x perché è una struttura poco complessa in detto microcontrollore. Nei microcontrollori e microprocessori dove lo Stack viene utilizzato anche per memorizzare altri dati esso assume un'importanza fondamentale, a causa delle aree variabili che vengono occupate.

ne prelevando (3) dallo Stack l'indirizzo di ritorno (`0052`) e riportando lo SP in posizione di livello 2.

Effettuato il ritorno nella subroutine SUB_1 il micro si deve preparare ad eseguire il ritorno nel `main`, prelevando (4) dallo Stack l'indirizzo `0013` e spostando lo *Stack Pointer* in posizione di livello 1.

Analizzando lo Stack in queste quattro fasi si nota come esso funziona effettivamente come una pila o catasta: gli indirizzi di ritorno vengono effettivamente impilati e prelevati secondo la regola del *First In Last Out* (**FILO**) che indica che la prima informazione impilata nella catasta è anche l'ultima ad essere prelevata.

La seconda serie di "fotografie" dello Stack serve a sottolineare come gli indirizzi di ritorno variano al variare dell'indirizzo di chiamata della subroutine.

Dopo aver eseguito la prima volta la routine SUB_1, si carica il valore $0x05$ nel registro di lavoro e si richiama una seconda volta la SUB_1. Ciò implica che si deve ora memorizzare (5) l'indirizzo di ritorno `0015` prima di saltare nuovamente nella SUB_1. Da questo momento in poi tutto procede come nella prima esecuzione della due subroutine, fino al prelievo (6) dell'indirizzo di ritorno al `main` che rimanda all'ultima istruzione che, infine, azzera nuovamente la variabile $Pippo$.

Un esercizio che lo studente può svolgere utilmente consiste nel registrare l'evolversi del contenuto della variabile $Pippo$ durante l'esecuzione delle istruzioni, verificando che le subroutine, pur eseguendo sempre le stesse istruzioni, trattano, invece, valori diversi di detta variabile.

Rimangono da esaminare tre aspetti fondamentali della programmazione che coinvolgono lo Stack:

- la limitata "profondità" dello Stack a soli 8 livelli;
- la necessità di memorizzare anche altre informazioni oltre l'indirizzo di ritorno;
- lo "sconfinamento" del Program Counter in altre pagine di memoria durante l'esecuzione delle subroutine.

La **limitata profondità** dello Stack implica che il programmatore deve porre estrema attenzione alla *nidificazione* delle chiamate a subroutine, dove per nidificazione delle chiamate si intende il richiamo di una subroutine all'interno di un'altra subroutine. Tale processo può essere effettuato per un massimo di 8 livelli.

Quest'ultima informazione va presa alla lettera: nel caso si preveda l'uso delle interruzioni (argomento che verrà trattato nei prossimi paragrafi) si deve tenere a mente che l'interrupt è asincrono, quindi non è possibile stabilire l'esatto momento in cui verrà scatenato tale evento. Ciò significa che l'effettivo livello di nidificazione è 7, dato che l'ottavo non è determinabile a livello temporale.

La **necessità di memorizzare altre informazioni** nasce proprio dalla presenza di eventuali interruzioni. Quasi sempre, infatti, le routine di interruzione (*Interrupt Service Routine* - **ISR**) "sporcano" (ossia modificano) alcuni dati, ad esempio il contenuto del registro W e il registro di stato.

Se detti registri non vengono adeguatamente salvati prima di eseguire la ISR e ripristinati prima di uscire dall'interruzione possono verificarsi situazioni potenzialmente assai dannose.

A titolo esemplificativo si valuti il codice riportato in fig. 1.10.

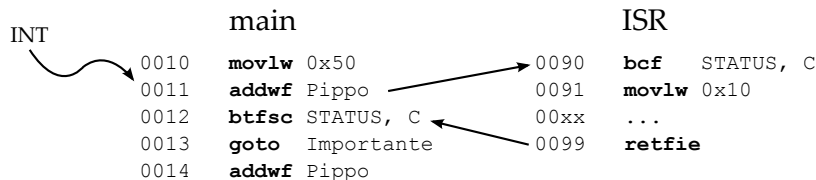



Figura 1.10: ISR senza salvataggio del registro di stato

La prima istruzione del main permette il caricamento del registro di lavoro con il valore $0x50$. Durante l'esecuzione della seconda istruzione (che viene comunque eseguita) viene rilevato un'interruzione da parte del micro. In tal caso il normale flusso del programma deve essere interrotto e deve urgentemente essere eseguita la ISR.

Si noti, però, che l'istruzione di somma (`addwf Pippo`) potrebbe aver provocato un supero di capacità dell'ALU (il che significa che il risultato non è memorizzabile in soli 8 bit). In tal caso il registro di stato **setta il flag di carry** (traduzione: pone a 1 il bit di riporto) in modo che sia notificato il supero di capacità.

L'istruzione seguente viene eseguita nell'ISR, azzerando il flag di carry e, l'istruzione seguente, modificando il registro W. Naturalmente si deve supporre che dette istruzioni siano effettivamente necessarie e funzionali. Purtroppo, però, sono funzionali per la routine di interruzione, ma non per il main, dato che sono state distrutte due informazioni fondamentali: il contenuto del registro W e parte del contenuto del registro STATUS.

Quando si rientra nel main, l'istruzione all'indirizzo `0012` testa il bit di carry nel registro di stato **e lo trova a 0 indipendentemente dall'esito della somma precedente**. Ciò provoca lo *skip* del `goto` (viene cioè ignorata l'istruzione di `goto` per eseguire direttamente la successiva) e non si salta all'indirizzo `Importante`, creando un sicuro danno.

Tali situazioni possono essere facilmente risolte salvando i registri coinvolti. L'argomento verrà trattato nei prossimi paragrafi.

Altra possibile fonte di guai è data dallo **"sconfinamento" del PC in altre pagine di programma**. Il main potrebbe, infatti chiamare chiamare delle subroutine che sono poste in altre pagine di memoria da quella chiamante. Ciò implica la preventiva modifica dei 2 bit più significativi dell'indirizzo, dato che l'istruzione di `call` fornisce solo i primi 11 bit di indirizzo.

Tale azione, ovvero la selezione della pagina di memoria di destinazione, richiede molto ordine da parte del programmatore e qualche accorgimento che faciliti detta procedura.

Questi verranno analizzati nei prossimi paragrafi. Per ora è sufficiente aver introdotto l'argomento, in modo che parte di esso venga assimilato dallo studente.

1.7.2 I vettori di salto

Nel paragrafo 1.7 si era sottolineata l'importanza delle prime 5 locazioni di memoria della pagina 0. In particolare 2 locazioni di memoria rivestono particolare importanza: l'indirizzo `0000` e l'indirizzo `0004`. Queste locazioni sono chiamate rispettivamente **vettore di reset** (*Reset Vector*) e **vettore di interruzione** (*Interrupt Vector*).

In particolari circostanze il microcontrollore inizia (o ricomincia) l'esecuzione del programma a partire dalla locazione di memoria `0000`, ovvero dalla prima.

Ciò avviene sostanzialmente in tre diverse occazioni:

- allo *startup*, ovvero all'accensione del micro, cioè quando esso viene alimentato;
- in seguito ad un segnale di *Reset* sul piedino $\overline{MCLR}$ del micro;
- in seguito ad un intervento del *Watch Dog Timer* (vedi cap. 1.10.3).

Quando uno di questi tre eventi ha luogo, il micro esegue le istruzioni a partire dalla locazione `0000`. Il programmatore è libero di gestirsi le prime istruzioni come preferisce, ma solitamente la prima istruzione è semplicemente quella indicata nel sottostante codice:

Listing 1.6: Vettore di Reset

```
1. 0000          goto Inizio     ;Salta all'inizio del programma
```

Una funzione del tutto analoga è svolta dal vettore di interruzione. Quando una periferica o un evento esterno attiva la procedura di interruzione, il micro salta alla locazione `0004`. In tale posizione viene normalmente posto un'istruzione di salto, come indicato nel sottostante codice:

Listing 1.7: Vettore di Reset

```
1. 0000          goto Inizio     ;Salta all'inizio del programma
2. 0001
3. 0002
4. 0003
5. 0004          goto ISR        ;Salta all'ISR
6. 0005 Inizio: ...              ;Prima istruzione del programma
   ...
X. xxxx ISR:    ...              ;Inizio ISR
```

1.8 Il Clock

Una macchina sequenziale non può funzionare senza un apposito segnale di clock. Si tratta di un segnale digitale a frequenza elevata avente il compito di "scandire il tempo" al micro. I microcontrollori della famiglia PIC16F87x accettano una frequenza di clock massima di 20MHz, mentre non c'è limite pratico alla frequenza operativa minima.

Siccome, però, la frequenza di clock e la tensione di alimentazione influiscono fortemente il consumo del microcontrollore, esistono varie versioni che limitano la frequenza massima a 4, 10, 16 o 20MHz. Se si prevede un funzionamento a batterie del dispositivo e se la velocità di esecuzione delle istruzioni non costituisce un problema insormontabile, conviene studiare attentamente il *datasheet* al fine di individuare il micro che meglio si adatta alle prestazioni richieste.

I blocchi interni al micro responsabili della generazione del clock sono evidenziati in fig. 1.11.

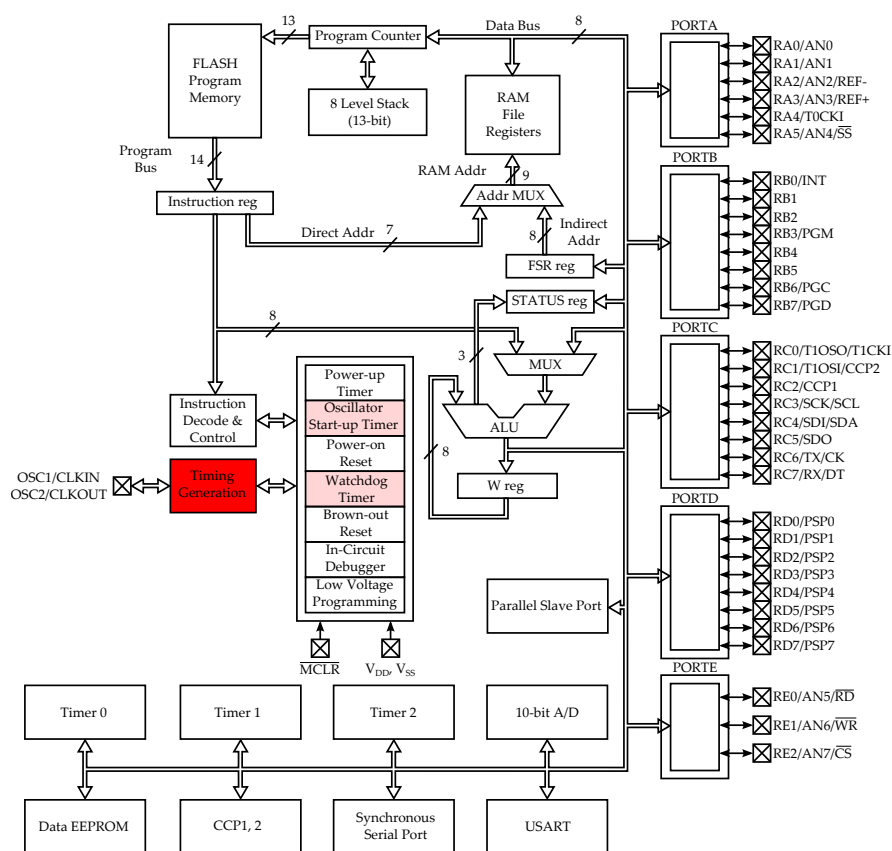



Figura 1.11: La generazione del clock

La prima caratteristica che un segnale di clock deve avere è la stabilità. Molti processi fondano la loro efficienza sulla precisione con la quale il micro misura il tempo. In tali casi è fondamentale che il segnale di clock, che scandisce ogni attività della macchina sequenziale, sia assolutamente stabile nel tempo, in modo che non vi siano variazioni nella frequenza di clock.

Sostanzialmente esistono quattro modi diversi per ottenere il suddetto segnale:

- mediante quarzo esterno;
- mediante risuonatore esterno;

- mediante rete RC esterna;
- mediante rete RC interna.

La prima soluzione offre un'ottima stabilità della frequenza di clock nel tempo anche in presenza di variazioni di temperatura. Non è una soluzione costosa nè eccessivamente ingombrante, anche se non si potrà mai parlare di miniaturizzazione del quarzo.

La soluzione che prevede il risuonatore offre una qualità leggermente inferiore del segnale, costi minori ed ingombri paragonabili. Si tratta di una soluzione molto diffusa là dove non sono necessarie grandi stabilità al variare della temperatura.

La rete RC esterna offre una stabilità in frequenza molto bassa e non può essere utilizzata in quei sistemi ove la misurazione del tempo deve essere precisa. Si tratta, però, di una soluzione a costo ed ingombro molto bassi, date le dimensioni dei due componenti che formano la rete RC.

La rete RC interna offre una soluzione già integrata che fornisce al micro un segnale di clock alla frequenza di 8MHz (divisibile, come si vedrà, mediante il prescaler interno. Si tratta, ovviamente, della più economica delle soluzioni, essendo essa a costo zero, a fronte di una stabilità di frequenza piuttosto bassa.

Questa soluzione offre un secondo vantaggio immediato: il risparmio di spazio sul circuito stampato, data l'assenza di componenti esterni.

Il progettista deve scegliere una delle seguenti quattro modalità di produzione del segnale di clock:

- LP: mediante quarzo a bassa frequenza e a bassa dissipazione di potenza;
- XT: mediante quarzo o risuonatore a media frequenza;
- HS: mediante quarzo o risuonatore ad alta frequenza;
- RC: mediante rete RC.

Nei primi tre casi (utilizzo del quarzo o del risuonatore) il costruttore consiglia il seguente scema al progettista:

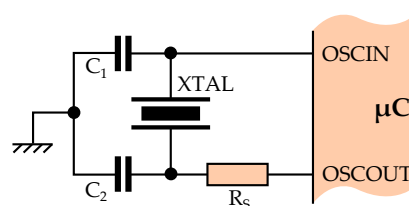



Figura 1.12: Circuito di clock con quarzo

Il componente XTAL può essere sia un quarzo che un risuonatore ceramico. In quest'ultimo caso si ponga attenzione che alcuni risuonatori posseggono già al loro interno, i due condensatori C_1 e C_2 da collegare verso massa, per cui non è necessario utilizzare i condensatori esterni. Detti risuonatori sono facilmente riconoscibili perché dotati di tre pin anziché due. Il pin centrale va connesso a massa.

La soluzione del risuonatore con i condensatori integrati non è da disdegnare, perché i due condensatori sono già correttamente dimensionati in base

alla frequenza di oscillazione ed offrono, quindi, un'alta garanzia di funzionamento.

Se, invece, si utilizza un quarzo o un risuonatore senza i condensatori integrati, il costruttore, in base alla frequenza di oscillazione voluta, consiglia i seguenti valori di C_1 e C_2:

Osc Type	Frequency	C1 range	C2 range
LP	32 kHz	33pF	33pF
LP	200 kHz	15pF	15pF
XT	200 kHz	47-68pF	47-68pF
XT	1 MHz	15pF	15pF
XT	4 MHz	15pF	15pF
HS	4 MHz	15pF	15pF
HS	8 MHz	15-33pF	15-33pF
HS	20 MHz	15-33pF	15-33pF

Tabella 1.2: Valori consigliati di C_1 e C_2

Il valore del resistore R_S non è normalmente fornito dal costruttore. Il resistore si rende necessario solamente per quarzi di tipo AT e vale circa un centinaio di Ohm. Negli altri casi il resistore deve essere cortocircuitato.

Nel caso, invece, in cui il progettista opti per la soluzione RC, il costruttore indica il seguente schema:

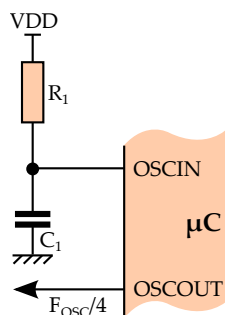



Figura 1.13: Circuito di clock con rete RC

I valori consigliati per il condensatore ed il resistore sono i seguenti:

$$C_1 > 20pF \qquad 3k\Omega < R_1 < 100k\Omega$$

In tal caso il microcontrollore fornisce nella propria uscita `OSCOUT` la frequenza di oscillazione divisa per 4, equivalente ad un ciclo macchina (vedi il paragrafo 1.5).

E' anche possibile pilotare il micro con un oscillatore già pronto. In tal caso è sufficiente rispettare la massima frequenza di oscillazione del microcontrollore e fornire detto segnale all'ingresso `OSCIN`.

Si richiama ancora l'attenzione dello studente sul consumo di energia. Per quelle applicazioni che sono alimentate a batteria è necessario diminuire via via la frequenza di clock fino a soddisfare le esigenze di durata della fonte di alimentazione. In molti casi si è costretti a scegliere la frequenza di oscillazione più bassa, ossia i 32568Hz.

1.9 Il Reset

Una macchina sequenziale necessita di un segnale di reset, simile a quello di fig. 1.14, che ponga il sistema in uno stato noto all'accensione.

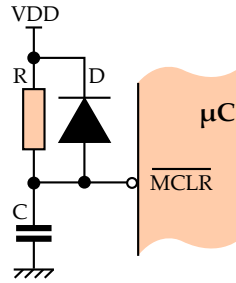



Figura 1.14: Circuito di reset

Il segnale di reset, normalmente attivo basso, pone il micro in uno stato noto ("pronto a partire"). Oltre ad inizializzare tutte le periferiche, **azzera il PC**, in modo che, dopo il reset, venga eseguita l'istruzione posta all'indirizzo 0000*h*, ovvero nel Vettore di Reset (vedi il paragrafo 1.7 ed la figura 1.7).

Affinché tale processo, che avviene all'accensione, si svolga correttamente si devono dimensionare in maniera oculata i valori di R e di C. Tale dimensionamento è eseguito dal progettista.

Prima di illustrare il dimensionamento di detti valori si deve, però, aprire una breve parentesi.

Dalla fisica sappiamo che quando si alimenta con un segnale a gradino una rete RC, supponendo il condensatore inizialmente scarico, questo si carica con legge esponenziale asintotica, come indicato in figura 1.15:

La legge di carica è data dalla seguente relazione:

$$V_C = V_{DD}\left(1 - e^{-\frac{t}{RC}}\right) \tag{1.2}$$

La tensione sul condensatore raggiunge il valore $\overline{V_{RES}} = 2/3V_{DD}$ dopo un tempo pari a τ ed il 98% del valore asintotico finale (ossia V_{DD}) dopo un tempo 3τ, con $\tau = RC$ espresso in secondi.

Quindi la 1.2 può essere riscritta nel seguente modo:

$$\frac{V_{DD} - V_C}{V_{DD}} = e^{-\frac{t}{\tau}} \tag{1.3}$$

Applicando i logaritmi (che lo studente del terzo corso potrebbe ancora non conoscere):

$$-\frac{t}{\tau} = ln\frac{V_{DD} - V_C}{V_{DD}} \quad \Leftrightarrow \quad -\frac{t}{\tau} = ln\frac{1}{3} \tag{1.4}$$

e siccome il membro di destra equivale, circa, a -1, si ha:

$$t \approx \tau \tag{1.5}$$

Ora lo studente ha tutti gli elementi per poter determinare il valore di R e C secondo le proprie necessità.

Il valore $\overline{V_{RES}}$ è quel valore di tensione, posta all'ingresso $\overline{MCLR}$ del micro, che lo fa uscire dallo stato di reset. Ciò significa che τ rappresenta il **tempo di reset**. Normalmente il progettista sceglie un tempo di reset di qualche millisecondo, ad esempio $10ms$. In tal caso, imponendo il valore di $R = 10k\Omega$ si calcola facilmente il valore di C:

$$t \approx RC \quad \Leftrightarrow \quad C = \frac{10^{-2}}{10^4} = 10^{-6} = 1\mu F \tag{1.6}$$

Rimane da definire il ruolo del diodo. Anche in questo caso lo studente del terzo corso potrebbe trovarsi in imbarazzo di fronte ad un componente che potrebbe non aver ancora studiato dettagliatamente. Saranno sufficienti pochi concetti aggiuntivi illustrati brevemente per spiegarne il ruolo all'interno della rete di fig. 1.14.

Quando viene a mancare la tensione di alimentazione (che in fig. 1.16 per semplicità si è supposta passare da V_{DD} a 0 in un tempo pari a zero), il condensatore si scarica con andamento esponenziale, quindi piuttosto lentamente. Questo comporta che il microcontrollore non risulti alimentato, mentre rimane alimentato il solo ingresso $\overline{MCLR}$, il che è vivamente sconsigliato dal costruttore. Introdurre un diodo come indicato in figura 1.14 permette al condensatore di scaricarsi velocemente attraverso la resistenza differenziale del diodo. L'andamento della scarica è illustrato in fig. 1.16 con linea tratteggiata.

Detta resistenza differenziale va a porsi in parallelo ad R, portando il valore ohmico del parallelo da $10k\Omega$ a $4-5\Omega$. In tal modo il pin $\overline{MCLR}$ non risulta essere praticamente alimentato quando non lo è il microcontrollore.

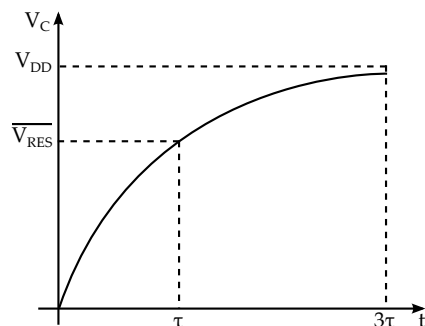



Figura 1.15: Curva di carica di un condensatore

Lo studente che non possiede ancora le necessarie nozioni per comprendere completamente il modo in cui il diodo opera, ritenga sufficiente quanto appena esposto.

Lo stato di reset non viene provocato solamente dalla rete RC esterna, bensì anche dai seguenti eventi:

- *Power-on Reset* (**POR**);
- reset di $\overline{MCLR}$ durante lo *sleep*;
- reset per intervento del **WDT** durante il normale funzionamento;
- reset per intervento del **WDT** durante lo *sleep*;
- *Brown-out Reset* (**BOR**).

Analizziamoli uno ad uno.

1.9.1 Il Power-on Reset

Ogni qual volta viene rilevato un passaggio per un valore compreso fra 1.2V e 1.7V del valore della tensione di alimentazione, viene generato un impulso di POR. Ciò implica, però, che il passaggio da 1.2V a 1.7V della tensione di alimentazione avvenga entro un tempo massimo. I datasheet indicano detto passaggio in 0.05V/ms, il che significa che la tensione di alimentazione del micro deve passare da 0V a 5V in un tempo minore o uguale a 100ms.

La figura 1.17 esplica quanto appena detto.

Finchè la tensione è inferiore a $1.2 \div 1.7V$ (detta tensione è indicata con POR_{TPR} nei datasheet, ma non è una tensione garantita dal costruttore) il segnale di POR è mantenuto attivo dal micro, che quindi a tutti gli effetti è in

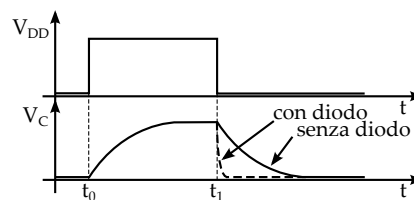



Figura 1.16: Scarica con diodo

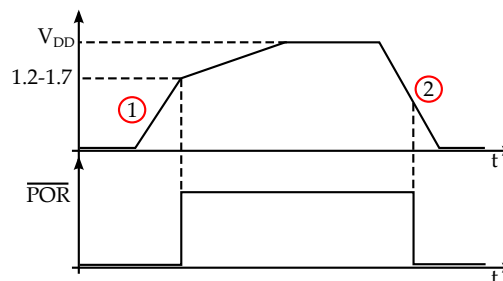



Figura 1.17: Segnale di POR

stato di reset. Se la tensione cresce (vedi il tratto indicato con ① in figura 1.17) con una pendenza superiore a 0.05V/ms **e** supera il valore di POR_{TPR}, allora il segnale di POR diventa disattivo ed il micro esce dallo stato di reset.

Il segnale di POR ridiventa attivo quando la tensione di alimentazione del micro scende sotto un valore (vedi punto indicato con ② in figura 1.17) che dipende dal micro: in realtà, però, anche in questo caso il costruttore non garantisce alcun valore certo per tale livello di tensione.

Per maggiori dettagli si veda PALMER [4].

1.9.2 $\overline{MCLR}$ durante lo *sleep*

Lo *sleep* rappresenta la modalità di risparmio energetico del micro. Da tale modalità il microcontrollore può uscire in vari modi, uno dei quali è appunto il rilevamento di un segnale di $\overline{MCLR}$.

La particolarità di tale "risveglio" è che a tutti gli effetti detto evento viene considerato come un normale reset: Il microcontrollore azzera il *program counter* e viene caricato (o se si preferisce: eseguito) il vettore di reset alla locazione di memoria 0000h.

Vi è però modo di distinguere un reset da $\overline{MCLR}$ durante uno *sleep* e durante la normale esecuzione di un'istruzione: a ciò sono preposti due bit del registro di *Status*. Testando a inizio programma detti bit si può sapere se il reset è avvenuto durante uno *sleep* o meno, e valutare, quindi, se il reset aveva il compito di "svegliare" il micro oppure no.

1.9.3 Intervento del WDT

L'acronimo WDT sta per *Watch Dog Timer*, di cui si parlerà tra poco nel paragrafo 1.10.3. Esso è un dispositivo avente il compito di avvertire il micro che un tempo preimpostato è scaduto e che ciò deve costituire allarme. Anche in questo caso viene lanciato un reset ed il micro viene reindirizzato, attraverso il vettore di reset, all'inizio del programma.

Come per quanto detto a proposito dello *sleep*, anche in questo caso è possibile valutare, attraverso il registro di stato, se il reset è stato generato dal WDT oppure no. In caso affermativo, il programma deve gestire un'emergenza, nata dalla presenza di un allarme.

E' compito del programmatore, in tal caso, valutare tutte le possibili emergenze al fine di determinare quale di esse si è verificata.

1.9.4 Intervento del WDT durante lo *sleep*

Il microcontrollore può essere svegliato dallo *sleep* anche dal WDT. In questo caso non si tratta di un allarme vero e proprio: non si tratta, cioè, della notifica di un malfunzionamento del sistema, ma semplicemente l'uscita da uno stato particolare del micro.

Detta uscita è voluta e programmata per cui non costituisce l'infrazione di un protocollo o di una procedura e va gestita, quindi, in maniera diversa da quella di un allarme vero e proprio.

Anche in questo caso il registro di stato contiene tutte le informazioni necessarie alla corretta valutazione del caso.

1.9.5 Il *Brown-out Reset*

Il microcontrollore deve operare in condizioni di tensione di alimentazione comprese fra una V_{DDmin} e una V_{DDmax}. Al di sotto della V_{DDmin} non si è sicuri che la memoria venga letta o scritta correttamente.

L'evento per cui viene a mancare la garanzia che tutte le periferiche e la memoria siano alimentate con tensione minima è detto *Brown-out*. Tale situazione è potenzialmente disastrosa perché difficilmente rilevabile dal micro stesso. Una periferica potrebbe scambiare un ingresso per un'uscita oppure una istruzione potrebbe essere mal decodificata, con effetti imprevedibili sul sistema.

Due sono le soluzioni per rilevare tale situazioni: sfruttare il circuito di *Brown-out* interno oppure aggiungere un circuito di *Brown-out* esterno.

La prima soluzione è integrata nel micro ed offre uns soluzione garantita dal costruttore, per cui risulta essere sicuramente preferibile. Il circuito interno rileva una qualsiasi tensione minore di V_{BOR} (circa 4V) per un tempo superiore a T_{BOR} (circa 100μs). Se tale situazione viene rilevata viene generato un reset da *Brown-out*.

Una volta che detto segnale è stato generato, il micro rimane in stato di reset finché la tensione di alimentazione non torna a superare il valore indicato da V_{BOR}. Quando ciò avviene il micro rimane in stato di reset ancora per un tempo pari a T_{PWRT} (circa 27ms). Se la tensione ritorna sotto il valore V_{BOR} durante il tempo T_{PWRT} il processo di *Brown-out* ricomincia da capo.

In alternativa al circuito interno (che può essere disabilitato durante la configurazione del micro) si può rilevare il *Brown-out* mediante un circuito esterno. Un esempio di detto circuito viene fornito direttamente dal costruttore, come indicato in figura 1.18.

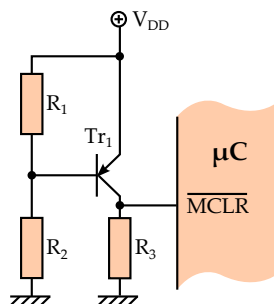



Figura 1.18: Circuito di Brown-out

Il circuito è piuttosto semplice: supponendo un transistor con h_{FE} elevata (>100), e scegliendo una coppia R_1 e R_2 adeguata (ossia tale che la corrente che le attraversa sia molto maggiore della corrente di base), si può ipotizzare che i suddetti resistori formino un partitore di tensione.

Il partitore va dimensionato in modo tale che, in funzionamento normale, il transistor sia in conduzione. In tal modo la tensione sull'ingresso di reset sarà sufficientemente alta da mantenere il micro in stato di reset inattivo.

Se la tensione V_{DD} cala, ai capi di R_1 non è più garantita la tensione di V_{EBon} e il transistor si interdice. In tal caso non circola più corrente in R_3 e la tensione sull'ingresso di reset diventa pari a zero volt.

Il dimensionamento dei resistori R_1, R_2 e R_3 si effettua, grossolanamente, nel seguente modo (senza tener conto delle tolleranze dei componenti):

$$\begin{cases} V_{EBon} = V_{DD}\frac{R_1}{R_1+R_2} \\ I_{R1} \gtrsim 1mA \end{cases} \tag{1.7}$$

dove V_{EBon} vale tipicamente $0.7V$. Da ciò si evince che un possibile valore di R_1 è $R_1 = 680\Omega$, per cui, supponendo $V_{DD} = 5V$, si ha:

$$R_2 = \frac{(V_{DD} - V_{EBon}) \cdot R_1}{V_{EBon}} = \frac{(5-0.7)\cdot 680}{0.7} = 4177 \quad \Rightarrow \quad 3K9\Omega \tag{1.8}$$

Con detti valori si ha una $V_{EBon} = 0.85V$, ovvero sufficiente a garantire la saturazione del transistor. Inoltre, si ha:

$$I_{R1} + I_B = I_{R2} \quad \Leftrightarrow \quad I_B = \frac{V_{DD} - V_{EBon}}{R_2} - \frac{V_{EBon}}{R_1} = 73\mu A \tag{1.9}$$

il che significa una corrente di collettore pari a $I_C = I_3 = h_{FE}I_B \simeq 7mA$, se si ipotizza $h_{FE} = 100$.

Supponendo che si abbia l'interdizione del transistor per $V_{EBon} = 0.65V$, ciò avviene per $V_{DD} = 3.8V$, che rappresenta, in questo caso, la tensione di *Brown-out* del micro.

Il valore di R_3 deve garantire il corretto valore di tensione all'ingresso $\overline{MCLR}$, quando il transistor conduce, ossia un valore maggiore di $0.8 \cdot V_{DD}$:

$$V_{\overline{MCLR}off} = 0.8 \cdot V_{DD} < R_3 I_3 \tag{1.10}$$

per cui

$$R_3 > \frac{0.8 \cdot V_{DD}}{I_3} > 571\Omega \quad \Rightarrow \quad 1K\Omega \tag{1.11}$$

Il dimensionamento del circuito è stato fatto senza tener conto delle tolleranze delle reistenze. Ciò è inaccettabile in ambito non scolastico. Inoltre, se si desidera una tensione di *Brown-out* estremamente precisa è necessario introdurre un punto di taratura nel circuito. Ciò si effettua mediante un trimmer posto in serie a R_2, dopo aver ricalcolato alla luce di ciò il valore di detta resistenza.

1.10 Le periferiche di sistema

Il microcontrollore è dotato di alcune periferiche che hanno lo scopo di migliorarne l'affidabilità e l'efficienza. Non sono da considerarsi periferiche ad "uso utente" come quelle che verranno illustrate nei prossimi paragrafi (convertitore analogico digitale, *timer*, periferiche di I/O, ecc.).

Tali periferiche sono evidenziate in figura 1.19.

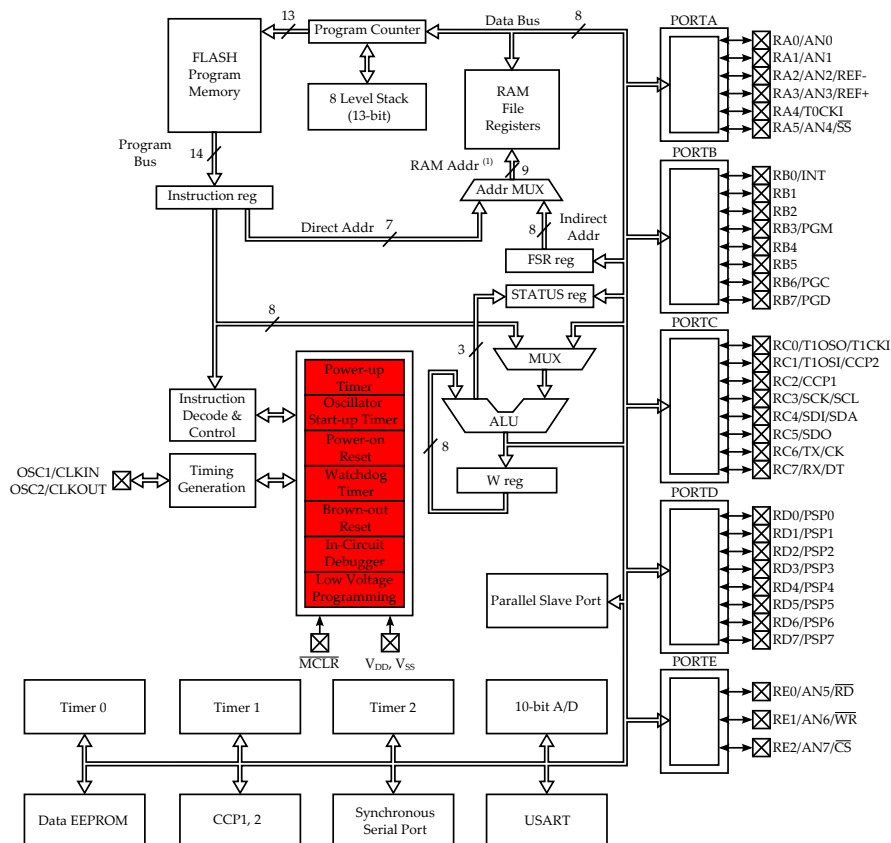



Figura 1.19: System Peripherals

Alcune di esse sono già state introdotte brevemente. Nei prossimi paragrafi verranno trattati esaustivamente il *Power-up Timer*, l'*Oscillator Start-up Timer*, il *Watchdog Timer*, l'*In-Circuit Debugger* e la periferica che permette il *Low Voltage Programming*.

1.10.1 Il *Power-up Timer*

Il *Power-up Timer* è sostanzialmente un temporizzatore avente il compito di mantenere il microcontrollore in stato di reset finché non si è stabilizzata la tensione di alimentazione e le singole parti che compongono il dispositivo non sono perfettamente funzionanti.

Dopo che il circuito di *Brown-out* ha rilevato che la tensione di alimentazione ha superato il valore V_{BOR} (vedi ① della temporizzazione di fig.1.20), viene atteso un tempo pari a T_{BOR} (vedi ②) dopodiché viene lanciato il *Power-up Timer* con il tempo T_{PWRT} (vedi ③). Dopo tale tempo si ritiene che il microcontrollore sia alimentato correttamente.

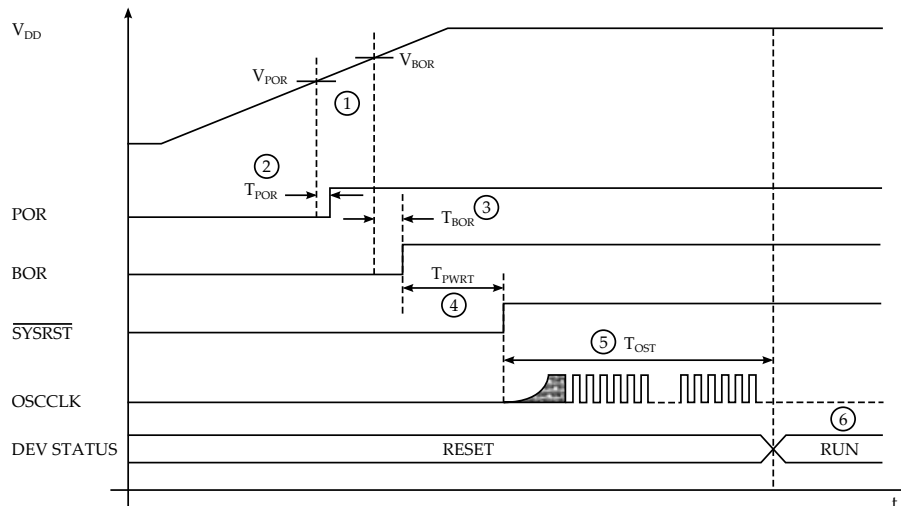



Figura 1.20: Temporizzazione di reset

Il tempo T_{PWRT} ha valore nominale di $72ms$. Si noti che dopo tale tempo il microcontrollore non è ancora operativo, dato che si deve attendere il tempo di assestamento dell'oscillatore, come evidenziato nel prossimo paragrafo.

1.10.2 L'*Oscillator Start-up Timer*

L'*Oscillator Start-up Timer* è un timer avente il compito di attendere che l'oscillatore si sia stabilizzato ed oscilli alla frequenza nominale. Il tempo T_{OST} (vedi ⑤ di fig. 1.20) è costante e pari a 1024 cicli di clock a partire da quando l'oscillatore ha raggiunto la frequenza nominale di oscillazione.

Passato tale tempo, il microcontrollore è perfettamente funzionante e pronto ad eseguire il salto indicato dal vettore di reset.

Si noti che l'*Oscillator Start-up Timer* viene attivato dal micro solamente in seguito ad uno *start-up* e non in seguito ad un semplice reset.

1.10.3 Il *Watchdog Timer*

Il *Watchdog Timer* è un temporizzatore avente funzione di "cane da guardia" (*Watch Dog*). Si tratta di un dispositivo estremamente importante nell'economia di una macchina sequenziale, per cui si ritiene utile proporre una tratta-

zione di tipo generale e non unicamente relativa al microptocessore oggetto dei presenti appunti.

La necessità di un "cane da guardia" nasce da un problema sempre presente nei sistemi di controllo: il malfunzionamento dell'hardware e/o il disturbo irradiato o condotto[6].

Dal punto di vista teorico, un programma può essere scritto bene oppure male ma esegue sempre ciò che il programmatore ha scritto, che per il momento si suppone sensato e corretto. Ma non è sempre così.

Abbiamo visto nel sottoparagrafo 1.9.5, ad esempio, che per motivi non dipendenti dal progettista la tensione di alimentazione può improvvisamente scendere al di sotto di una soglia critica che garantisce il buon funzionamento della macchina microprogrammata.

In tale frangente il micro può, ad esempio, decodificare in maniera errata un'istruzione e saltare chissà dove invece di attivare il convertitore analogico-digitale. Un tale malfunzionamento, dovuto nell'esempio ad una caduta di tensione, può essere potenzialmente catastrofico sia per il sistema controllato che per l'uomo.

Ovviamente ciò non deve succedere.

Al fine di evitare tale problema è utile l'utilizzo del *Watchdog*. Esso è sostanzialmente un timer che viene continuamente ricaricato e che conta automaticamente in decremento.

La prima cosa da fare è quantificare nella maniera più precisa possibile la durata del *main loop*. A tal fine si suppone che il codice del software sia stato scritto in modo da evitare che vi siano delle *task* che introducano sostanziose variazioni nei tempi di esecuzione.

Prendendo come riferimento il diagramma degli stati di fig. 1.21 che illustra un esempio di *main loop* composto da 4 *task*, si suppone che la *task* 1 venga richiamata ciclicamente ogni $t_1 + t_2 + t_3 + t_4 = t_{loop} ms$ [7].

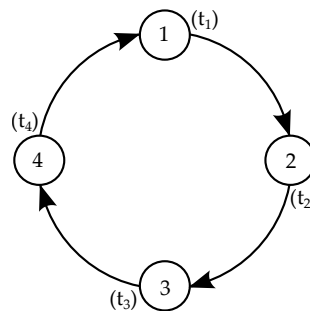



Figura 1.21: Main loop

[6]Il disturbo irradiato è generato da un dispositivo che funziona ad alta frequenza ed è veicolato da un'onda elettromagnetica che si concatena ad una qualsiasi "antenna" (ossia un qualsiasi filo o pista di circuito stampato avente una determinata lunghezza) presente sulla scheda elettronica. Il disturbo condotto è generato da un dispositivo tipicamente di potenza che veicola il disturbo attraverso la massa comune.

[7]In figura in tempi sono indicati tra parentesi, in modo che essi non vengano confusi con le variabili d'ingresso della macchina sequenziale.

Se la Δt_{loop} è nota e contenuta, si può impostare un tempo t_{wd} di *watchdog* pari a $fs \cdot (t_{loop} + \Delta t_{loop})$, dove fs è un fattore di sicurezza, ad esempio, pari a 1.2.

Ogni volta che la *task* 1 viene eseguita, viene anche ricaricato il WDT con il tempo t_{wd} e fatto decrementare, supponiamo, ogni μs. Siccome, in realtà, il ciclo viene complessivamente eseguito nel tempo $t_{loop} \pm \Delta t_{loop}$, il WDT non arriverà mai al valore zero, quindi non potrà mai lanciare un WDT Reset (vedi 1.9.3), notificando in tal modo, un malfunzionamento del sistema.

Se, però, per un qualche motivo, il WDT dovesse riuscire a raggiungere il valore zero prima di essere ricaricato con il valore t_{wd}, significherebbe che il tempo di esecuzione del ciclo si è protratto oltre il tempo normale e corretto: tale situazione va notificata perché potenzialmente foriera di malfunzionamento dell'hardware.

Il WDT notifica tale evento resettando il *Program Counter* ed obbligando il microcontrollore a saltare al vettore di *Reset*. Durante tale operazione il programma deve verificare, controllando il registro di *Status*, se il *Reset* è stato causato dal WDT o meno. In caso affermativo significherebbe che è stata rilevata un'anomalia e si dovrebbe dare inizio ad una procedura di messa in sicurezza del sistema.

1.10.4 L'*In-Circuit Debugger*

L'*In-Circuit Debugger* è una circuiteria che permette di eseguire il *debugging* del software (ovvero la ricerca e correzione delle anomalie del software) direttamente sulla scheda, senza l'aggiunta invasiva di *probes* o altra elettronica.

A tal fine viene posto un connettore RJ11 sulla scheda che permette la connessione alla scheda dell'emulatore Microchip. Il circuito elettrico che funge da interfaccia fra emulatore e micro è visualizzato in fig. 1.22.

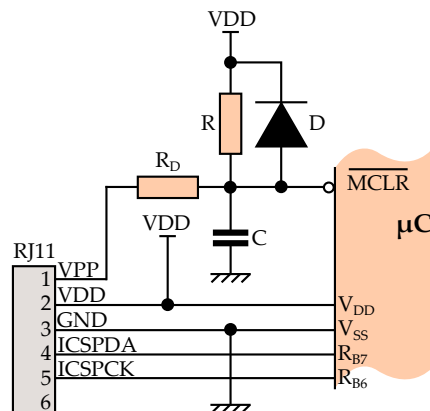



Figura 1.22: Interfaccia *In-Circuit Debugger*

L'emulatore verifica la presenza di alimentazione sulla scheda controllando la tensione presente sui pin 2 e 3 del connettore RJ11.

Il piedino 1 del connettore è collegato ad un resistore $R_D = 100\Omega$ che permette un eventuale reset non invasivo della scheda.

Il *debugging* vero e proprio avviene attraverso i pin R_{B6} e R_{B7} del micro, che sono a ciò dedicati. Ciò solleva un problema. Detti pin, durante il *debugging*, non possono essere dedicati ad altro. Durante il funzionamento normale, per contro, sono disponibili per il sistema. Non conviene, però, dedicare tali linee a compiti complessi, perché detti compiti non sono verificabili durante il collaudo. A parte i casi eccezionali, R_{B6} e R_{B7} sono solamente dedicati a funzioni di interfacciamento con l'emulatore Microchip.

In generale, le risorse sottratte dall'emulatore al sistema sono indicate nella sottostante tabella.

Risorsa	Descrizione
I/O pins	R_{B6}, R_{B7}
Stack	1 level
Program Memory	Address 0000h must be `NOP`
	Last 100h words
Data Memory	0x070 (0x0F0, 0x170, 0x1F0)
	0x1EB - 0x1EF

Tabella 1.3: Risorse utilizzate dall'emulatore

L'emulatore, oltre a R_{B6} e R_{B7} utilizza anche un livello dello stack, portando a 7 i livelli effettivamente utilizzabili dal programmatore.

Il vettore di reset deve contenere un'istruzione di `NOP` e deve essere seguita dal vettore vero e proprio (tipicamente un `GOTO Start`). Inoltre, le ultime 256 locazioni di memoria devono essere lasciate a disposizione dell'emulatore.

Infine, l'indirizzo $0x70$ della memoria dati (e quindi per sovrapposizione anche gli indirizzi $0xF0$, $0x170$ e $0x1F0$) deve essere lasciato libero, come pure il gruppo di indirizzi $0x1EB \div 0x1EF$.

Si noti, infine, che, attraverso lo stesso connettore, l'emulatore ha la possibilità di programmare la memoria flash con le istruzioni scritte dal programmatore.

Questo particolare introduce il prossimo sottoparagrafo.

1.10.5 Il *Low Voltage Programming*

Il *Low Voltage Programming* permette la programmazione della memoria flash attraverso la sola tensione di alimentazione presente sulla scheda. Questo permette la programmazione del micro senza la necessità di tensioni esterne ma usando solamente le risorse presenti sulla scheda.

Il LVP è abilitato sul micro di *default* e impegna il pin R_{B3}, che quindi non può essere usato come pin di I/O. Se vi è la necessità di utilizzare detto pin, si deve disabilitare il bit LVP nella parola di configurazione (vedi la sezione 3.10 per ulteriori dettagli).

Se, invece, il LVP è abilitato contemporaneamente all'abilitazione dei *pull-ups* sul PORTB, si deve azzerare il bit 3 di detto port affinché il dispositivo possa funzionare correttamente e senza conflitti.

1.11 Le periferiche

Il microcontrollore PIC16F877 possiede 9 periferiche, che si interfacciano verso l'esterno attraverso le linee del PORTA, PORTC, PORTD e PORTE, con esclusione del solo PORTB.

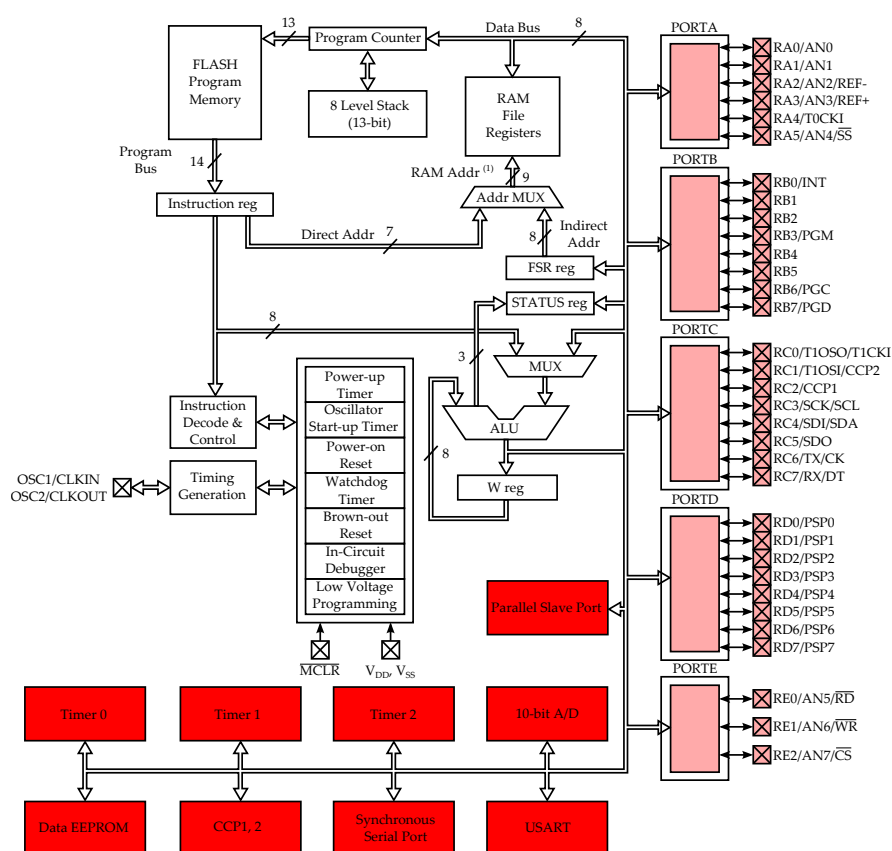



Figura 1.23: Le periferiche del PIC16F877

Il PORTB è comunque utilizzato dall'*In-Circuit Debugger*, per cui almeno due linee (R_{B6} e R_{B7}) sono dedicate allo scambio dati con l'emulatore.

Le periferiche sono le seguenti:

- 3 timer;
- 1 convertitore analogico-digitale da 10 bit e 8 canali;
- 1 memoria seriale di tipo Eeprom;
- 2 moduli CCP;

- 1 linea seriale sincrona;
- 1 linea seriale asincrona;
- 1 parallel slave port.

Dette periferiche verranno descritte nel dettaglio nei prossimi capitoli, per cui nel presente paragrafo ci si limiterà ad una sommaria descrizione che illustri grossolanamente la destinazione d’uso dei diversi dispositivi periferici.

1.11.1 I timer

Il microcontrollore PIC16F877 possiede 3 timer indipendenti. Essi permettono un conteggio preciso del tempo senza che sia necessario sotrarre eccessive risorse al micro, che rimane libero per altre attività. I timer hanno le seguenti caratteristiche principali:

Nome	Bit	Interruzione	Contatore
Timer 0	8	SI	SI
Timer 1	16	SI	SI
Timer 2	8	SI	NO

Tabella 1.4: Timers a disposizione del PIC16F877

Il **Timer 0** è un timer a 8 bit e può essere a sua volta abbinato ad un prescaler programmabile anch’esso a 8 bit. Il clock può essere sia interno che esterno. In questo secondo caso il timer può essere considerato un contatore di eventi esterni. A fine ciclo di conteggio (cioè nel passaggio da 0xFF a 0x00) il timer lancia interruzione, se abilitata. Se il micro è in stato di *Sleep*, l’interruzione lanciata dal timer 0 non lo sveglia.

Il **Timer 1** è un timer a 16 bit e può anch’esso funzionare sia da timer che da contatore di eventi esterni. Può essere abbinato ad un prescaler che divide ulteriormente per 2, 4 oppure 8 la sorgente del clock. Può lanciare interruzione a fine ciclo e tale interruzione, a differenza di quella lanciata dal Timer 0, sveglia il micro posto eventualmente in stato di *Sleep*.

Il **Timer 2** è un timer a 8 bit abbinato ad un prescaler e ad un postscaler. Lancia interruzione quando il valore raggiunto dal timer è uguale a quello memorizzato nel corrispondente *Period Register*. Non può funzionare da contatore di eventi esterni.

1.11.2 Il convertitore analogico-digitale

Il micro possiede un convertitore analogico-digitale da 10 bit. Il convertitore possiede 5 canali di conversione nei PIC16F873 e PIC16F876, mentre ne possiede 8 nei PIC16F874 e PIC16F877.

Il riferimento di tensione può essere sia esterno che interno ed il convertitore è in grado di lanciare interruzione alla fine della conversione.

Il dato a 10 bit è memorizzato su due distinti registri, per cui detto dato può essere *allineato a destra* o *allineato a sinistra*. Ciò significa che il convertitore è sostanzialmente in grado di fornire tre formati distinti del dato campionato:

- dato a 10 bit su due distinti registri;
- 8 bit meno significativi del dato su un unico registro;
- 8 bit più significativi del dato su un unico registro.

Il convertitore è in grado di convertitre segnali analogici anche in stato di *Sleep*. Tale situazione è consigliata dal costruttore se si devono campionare segnali con frequenza maggiore di 1MHz.

1.11.3 La Eeprom

I PIC16F873 e PIC16F874 possiedono 128 byte di Eeprom interna, mentre i PIC16F876 e PIC16F877 possiedono 256 byte di Eeprom interna.

La Eeprom è un tipo di memoria non volatile che trattiene, quindi, i dati ivi memorizzati anche nel caso in cui viene a mancare la tensione di alimentazione.

I tempi di scrittura in memoria sono quelli usuali delle Eeprom seriali: da 4ms tipici a 8ms massimi. Sia la lettura che la scrittura del dato richiede l'esecuzione di una procedura relativamente complessa, che verrà illustrata nel capitolo relativo alla Eeprom seriale.

Solitamente quest'area di memoria viene utilizzata per memorizzarvi i parametri di configurazione di un sistema, che possono essere modificati in sede di primo avviamento e collaudo del sistema.

1.11.4 I moduli CCP

Il PIC16F877 possiede due moduli CCP (*Capture/Compare/Pwm*) indipendenti, basati su due registri a 16 bit. Le tre tipologie di funzionamento sono le seguenti:

Capture Mode

Quando si verifica un determinato evento sul piedino CCP1, viene copiato il valore del Timer 1 nel registro a 16 bit del CCP e viene lanciata un'interruzione, se abilitata. Gli eventi possibili (e selezionabili) sul pin CPP1 sono i seguenti:

- ogni fronte di discesa presente sul pin CCP1;
- ogni fronte di salita presente sul pin CCP1;
- ogni 4 fronti di discesa presenti sul pin CCP1;
- ogni 4 fronti di salita presenti sul pin CCP1.

Compare Mode

Durante il funzionamento in modalità *Compare* il valore del registro di CCP viene costantemente confrontato con quello del Timer 1 e quando è verificata

l'uguaglianza dei due valori, oltre a lanciare interruzione, viene eseguita una delle seguenti azioni:

- la linea di CCP1 viene posta a 1 logico;
- la linea di CCP1 viene posta a 0 logico;
- la linea di CCP1 rimane nello stesso stato logico.

PWM Mode

PWM significa *Pulse Width Modulation*, ovvero modulazione a larghezza d'impulso. Si tratta di un modo consueto per pilotare motori in corrente continua o dispositivi che abbisognano di essere pilotati attraverso la variazione di *duty cicle* di un segnale. Tale argomento verrà ripreso dettagliatamente quando si approfondirà la periferica CCP.

Attraverso il registro di CCP ed il Timer 2 è possibile modificare, con una risoluzione di 10bit (con la quale si intende la effettiva risoluzione di un bit su un valore binario espresso su 10 bit), il *duty cicle* di un impulso ripetitivo con periodo impostato. Ciò permette un controllo in PWM, ad esempio di un motore, in maniera estremamente semplificata ed efficiente senza dover sprecare eccessive risorse del micro.

1.11.5 La linea seriale sincrona

La periferica di comunicazione seriale sincrona permette l'uso di due diversi standard di comunicazione seriale sincrona:

- Serial Peripheral Interface (SPI);
- Inter-Integrated Circuit (I^2C).

Entrambi gli standard permettono una comunicazione seriale sincrona del tipo Master-Slave fra dispositivi posti, possibilmente, sulla stessa scheda o, comunque, nelle immediate vicinanze del micro.

Si tratta di due standard estremamente diffusi e di larga applicazione. I dispositivi che usano detti standard sono solitamente le memorie Eeprom seriali, gli orologi/calendari, convertitori seriali, ecc.

1.11.6 La linea seriale asincrona

Il microcontrollore possiede una linea seriale asincrona, ovvero senza la presenza di un clock di sincronizzazione fra trasmittente e ricevente. La periferica che sovraintende a tale incombenza è detta USART (*Universal Synchronous Asynchronous Receiver Transmitter*). Essa può essere configurata in tre diversi modi:

- modalità asincrona (*full duplex*);
- modalità sincrona Master (*half duplex*);
- modalità sincrona Slave (*half duplex*).

L'USART utilizza una codifica di linea del tipo NRZ con baudrate selezionabile, un bit di star e un bit di stop. Non fornisce la diretta gestione del bit di parità che, però, può essere ottenuto per via software.

1.11.7 Il *Parallel Slave Port*

Il PSP è implementato solo nei PIC16F874 e PIC16F877. Esso permette l'interfacciamento verso dispositivi paralleli con parallelismo a 8 bit. La periferica è dotata di linee di abilitazione alla lettura/scrittura ($\overline{RD}$ e $\overline{WR}$). Tipicamente si tratta di memorie volatili di tipo parallelo con tempi di accesso piuttosto contenuti.

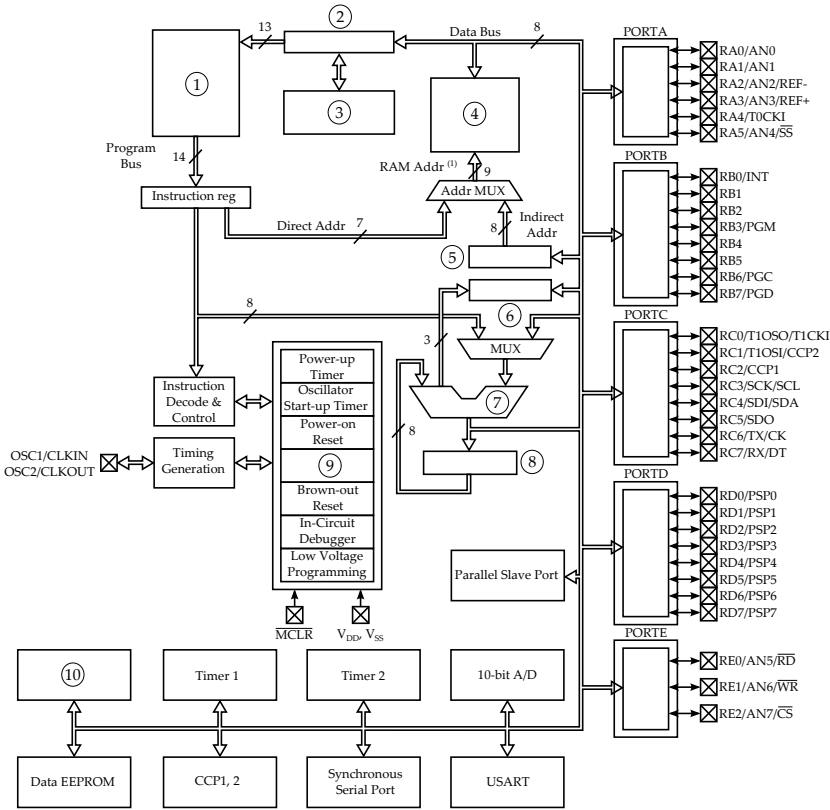
1.12 Esercizi

Gli esercizi riportati nelle seguenti pagine sono tutti relativi a quanto esposto nel capitolo 1. Le soluzioni degli esercizi sono riportate in Appendice A.

L'architettura del PIC16F877

1. ◊◊◊ Si elenchino le principali tipologie di macchine microprogrammate.
2. ◊◊◊ Quali sono gli ambiti di applicazione di ciascuna tipologia?
3. ◊◊◊ Si illustrino i diversi "tagli" della famiglia PIC16F87x in termini di memoria di programma e linee di I/O.
4. ◊◊◊ Quali sono le principali differenze fra le architetture di tipo Von Neumann e di tipo Harvard? Il PIC16F877 a quale delle due appartiene?
5. ◊◊◊ Si fornisca una generica definizione di "bus".
6. ◊◊◊ Cosa significano gli acronimi RISC e CISC?
7. ◊◊◊ Da cosa sono caratterizzati i micro di tipo RISC e CISC? Il PIC16F877 a quale delle due tipologie appartiene?
8. ◊◊◊ Qual è la funzione dell'ALU? A quanti bit di paralellismo opera?
9. ◊◊◊ In quali due registri (o gruppi di registri) può scrivere la ALU?
10. ◊◊◊ Quali sono i registri che forniscono i due operandi all'ALU?
11. ◊◊◊ Quale particolare registro viene aggiornato *sempre* dopo ogni operazione eseguita dall'ALU?
12. ◊◊◊ Qual è la funzione del Program Counter?
13. ◊◊◊ Qual è il valore massimo che può raggiungere il PC nel PIC16F877?
14. ◊◊◊ Si illustri il meccanismo di *pipelining*.
15. ◊◊◊ Qual è il vantaggio immediato del *pipelining*?
16. ◊◊◊ Cosa si intende con il termine *fetch*?
17. ◊◊◊ Cosa si intende con il termine *flush*?
18. ◊◊◊ Si illustri l'*indirizzamento diretto*.
19. ◊◊◊ Qual è la particolarità dell'*indirizzamento indiretto*?
20. ◊◊◊ Quando si usa tipicamente l'*indirizzamento indiretto*?
21. ◊◊◊ Qual è la particolarità dell'*indirizzamento relativo*?
22. ◊◊◊ Quando si usa tipicamente l'*indirizzamento relativo*?
23. ◊◊◊ Quanti e cosa sono i vettori di salto?
24. ◊◊◊ Qual è l'indirizzo del Vettore di Reset?

25. ◊◊◊ Qual è l'indirizzo del Vettore di Interruzione?
26. ◊◊◊ Perché si devono "salvare" i registri W e STATUS dopo un'interruzione?
27. ◊◊◊ Quali sono gli ultimi indirizzi delle diverse pagine della memoria di programma? Perché sono importanti?
28. ◊◊◊ Quante pagine di memoria di programma possiede il PIC16F877?
29. ◊◊◊ A cosa serve lo *Stack*?
30. ◊◊◊ Perché è bene non superare i 7 livelli di nidificazione delle chiamate a subroutine pur avendo lo *Stack* 8 livelli?
31. ◊◊◊ A cosa serve il segnale di clock?
32. ◊◊◊ Quanti e quali diversi modi per fornire il segnale di clock sono previsti dal micro?
33. ◊◊◊ Come si differenziano fra loro? Quale di essi è il più stabile e quale il più economico?
34. ◊◊◊ Si disegni e si dimensioni un possibile circuito di clock a 8MHz.
35. ◊◊◊ Qual è la massima frequenza di clock del PIC16F877?
36. ◊◊◊ Qual è la minima frequenza di clock del PIC16F877?
37. ◊◊◊ A cosa serve il segnale esterno di reset? A che pin è applicato?
38. ◊◊◊ Si disegni un possibile circuito di reset e si dimensionino R e C.
39. ◊◊◊ Si elenchino i diversi eventi che producono il reset del micro.
40. ◊◊◊ Si illustri il funzionamento del POR.
41. ◊◊◊ Cos'è il BOR e come funziona?
42. ◊◊◊ Si disegni un circuito esterno di *Brown-out* e lo si dimensioni.
43. ◊◊◊ Si illustri lo scopo del *Power-up Timer*.
44. ◊◊◊ Si illustri lo scopo dell'*Oscillator Start-up Timer*.
45. ◊◊◊ A cosa serve il segnale esterno di reset? A che pin è applicato?
46. ◊◊♦ Se il segnale di BOR diventa attivo dopo 50*ms* dal *power-up*, il micro dopo quanti *ms* dall'accensione esce dallo stato di reset?
47. ◊◊◊ Si illustri la funzionalità del WDT.
48. ◊◊◊ A cosa serve l'*In-Circuit Debugger*?
49. ◊◊♦ Quali risorse sottrae al programmatore?
50. ◊◊♦ Si disegni il circuito di interfaccia fra il micro e il connettore RJ11 dell'*In-Circuit Debugger*.

51. ◊◊◊ In quali casi è utile il circuito di *Low Voltage Programming*?
52. ◊◊◊ Quante e quali sono le periferiche del PIC16F877?
53. ◊◊◊ Quale periferica è mancante nei PIC16F876 e PIC16F873?
54. ◊◊◊ Quante sono complessivamente le linee di I/O del PIC16F877?
55. ◊◊♦ Nella figura sottostante è rappresentata una architettura parzialmente "muta" del microcontrollore PIC16F877. Sotto la figura sono indicate le diciture omesse in figura e numerate da 1 a 10. Si indichi la corretta numerazione a fianco delle diciture.

8 Level Stack	....
Timer 0	....
ALU	....
FSR reg	....
FLASH Program Memory	....
W reg	....
Program Counter	....
STATUS reg	....
Watchdog Timer	....
RAM File Registers	....

Capitolo 2

Le memorie del PIC16F877

Nel capitolo precedente si è accennato a due particolari memorie del PIC:

- la *FLASH Program Memory*;
- le *RAM File Registers*.

La prima è una memoria non volatile (in grado, quindi, di trattenere l'informazione in caso di caduta di tensione) che contiene il **programma** che il microcontrollore deve eseguire. Il programma è formato dalle singole istruzioni, che vengono scritte nella memoria o mediante un'apposito dispositivo esterno, chiamato **emulatore** oppure mediante un **programmatore**, anch'esso esterno.

La seconda è una memoria statica di tipo volatile, organizzata a **registri**, come tipico dei microcontrollori RISC. In questa memoria vengono memorizzate le variabili necessarie al programma per funzionare e le variabili necessarie al funzionamento del microcontrollore e delle relative periferiche.

Si è già accennato, nel capito precedente che tali memorie sono organizzate in maniera piuttosto particolare, per cui si dedicherà l'intero capitolo alla trattazione di tale argomento.

2.1 I *RAM File Registers*

Si è detto che la RAM è organizzata a registri. Ciò significa che ogni singola cella di memoria è trattata dal microcontrollore come un registro interno e non come una cella di memoria esterna. La principale differenza fra registro e cella di memoria consisten nel fatto che l'accesso al registro interno è più veloce e più generalizzato rispetto alla cella di memoria esterna.

Mentre il significato del termine “veloce” appare semplice ed immediato, il termine “generalizzato” merita un approfondimento.

Nei processori di tipo CISC esiste una netta distinzione fra la memoria (esterna o interna che sia) ed i registri. La prima è completamente dedicata all’utente, che può farne l’uso che preferisce. Normalmente la memoria RAM è la sede ove vengono memorizzate le varibili del programma in esecuzione.

I registri, invece, hanno ciascuno una destinazione d’uso ben precisa e svolgono una funzione chiaramente definita. Alcuni di tali registri svolgono un compito talmente importante che non è possibile accedervi direttamente da parte del programmatore. Ciò significa che il programmatore non ha facoltà di scrivere direttamente tali registri. Un esempio è il Program Counter, che è gestito direttamente dal micro e che non è accessibile nè in lettura nè in scrittura al programmatore. Questo perchè, secondo la filosofia CISC, si ritiene che il programmatore non debba alterare direttamente lo stato del Program Counter.

Nei microcontrollori di tipo RISC, invece, l’approccio è diverso (si veda la fig. 2.1).

□ Unimplemented data memory locations, read as '0'.
* Not a phisycal register.
Note 1: These registers are not implemented on the PIC16F876.
2: These registers are reserved, mantain these registers clear.

Figura 2.1: Data Memory Map

Si ritiene che tutti i registri abbiano le stesse caratteristiche e che non vi debbano essere differenze fra loro. Quindi si può modificare un qualsiasi registro indipendentemente dalla sua importanza. In tal modo molte operazioni vengono velocizzate. Lo svantaggio di tale filosofia e che si possono commettere errori grossolani in fase di scrittura del codice.

Da quanto appena detto emerge che i registri ad uso specifico e quelli ad uso generale sono equiparati. Infatti, pur divisi, sono accumunati nella stessa area di memoria. Detta area, per motivi tecnologici, è suddivisa in quattro banchi distinti, come indicato dal seguente specchietto:

Banco	Inizio	Fine
0	00h	7Fh
1	80h	FFh
2	100h	17Fh
3	180h	1FFh

Tabella 2.1: Indirizzi *Data Memory Map*

Siccome tale suddivisione in banchi può creare qualche problema in sede di scrittura del codice, si dedica il prossimo sottoparagrafo all'argomento.

2.1.1 I banchi RAM

La figura 2.1 a fronte e la tabella 2.1 evidenziano chiaramente che l'area di memoria della RAM è suddivisa in quattro parti uguali. Non emergono, però, le implicazioni, talvolta subdole, di tale suddivisione.

Per una loro corretta comprensione si deve brevemente introdurre il concetto e la sintassi della dichiarazione di variabile.

Quando il programmatore vuole dichiarare una variabile di tipo intero in linguaggio assembly utilizza la seguente sintassi:

Listing 2.1: Esempio di dichiarazione di variabili

```
VarB0   equ    0x20    ;Allocazione variabile in banco 0
VarB1   equ    0xA0    ;Allocazione variabile in banco 1
VarB2   equ    0x120   ;Allocazione variabile in banco 2
VarB3   equ    0x1A0   ;Allocazione variabile in banco 3

movf    VarB0, W       ;Carica in W il contenuto di VarB0
```

La variabile `VarB0` è *allocata* all'indirizzo 0x20 (oppure 20h, a seconda della notazione), il che significa che ad essa è riservato lo spazio opportuno (in questo caso 1 byte) in memoria ad un determinato indirizzo. La variabile `VarB1` è allocata all'indirizzo 0xA0, e così via.

Quando, però, la variabile `VarB0` viene, ad esempio, letta per porre il suo contenuto nel registro W (vedi l'ultima riga del listato 2.1) emerge un'ambiguità: l'indirizzo rappresentato dall'etichetta `VarB0` non viene usato nella sua

intierezza, non vengono, cioè, utilizzati tutti gli 8 bit dell'indirizzo per accedere al contenuto della variabile, ma solamente i 7 bit meno significativi. Tale scelta è determinata da motivi tecnologici e di architettura.

Mediante 7 bit si seleziona una qualsiasi locazione di memoria compresa fra 0x00 e 0x7F, che rappresenta esattamente l'area di un singolo banco di memoria. Rimangono, quindi, da definire ulteriori due bit per la selezione del banco di memoria.

Il meccanismo per accedere ad una qualsiasi delle variabili indirizzabili dal micro dipende dalla modalità di indirizzamento. L'uso dei 7 bit meno significativi è presente nell'indirizzamento diretto ed in quello indiretto.

2.1.2 I banchi nell'indirizzamento diretto

La selezione della locazione di memoria, nell'indirizzamento diretto, avviene come rappresentato nella figura 2.2.

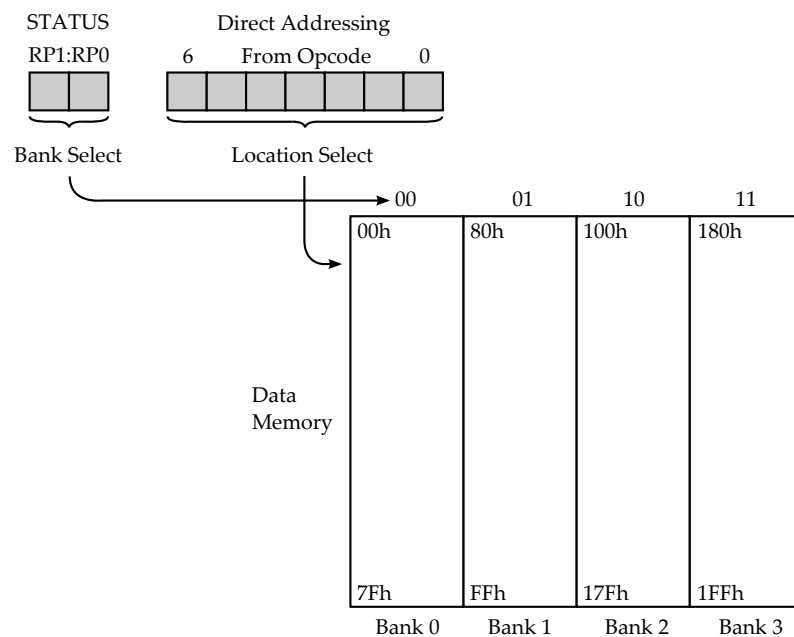



Figura 2.2: Indirizzamento diretto

A conferma di quanto detto si verifichi in fig. 1.6 a pagina 14 che effettivamente il bus degli indirizzi relativo all'indirizzamento diretto è largo soli 7 bit anziché 8.

Ciò significa che devono essere forniti ulteriori due bit per completare l'indirizzo, formato complessivamente da 9 bit. Tali bit vengono forniti dal registro di STATUS, attraverso i bit RP1 e RP0 (rispettivamente il bit 6 e 5 di detto registro).

La principale implicazione di tale modo di indirizzare le locazioni di memoria è evidenziato nel listato seguente.

Listing 2.2: Esempio di indirizzamento errato

```
;Seleziona il banco 0 della Data Memory
bcf     STATUS,RP1
bcf     STATUS,RP0

;Indirizzamento errato di VarB1, VarB2 e VarB3
movlw   0x00    ;Carica la costante 0 nel registro W
movwf   VarB0   ;Azzera correttamente VarB0
movwf   VarB1   ;Azzera nuovamente VarB0, non VarB1
movwf   VarB2   ;Azzera nuovamente VarB0, non VarB2
movwf   VarB3   ;Azzera nuovamente VarB0, non VarB3
```

Mediante le prime due istruzioni viene selezionato il banco 0 di memoria. Poi, mediante la terza istruzione, si carica il valore 0 nel registro W e poi, nelle istruzioni seguenti, si trasferisce il valore posto in W rispettivamente all'indirizzo indicato da `VarB0`, `VarB1`, `VarB2` e `VarB3`. Però, solamente il primo trasferimento va a buon fine, dato che i successivi tre operano sempre allo stesso indirizzo 20h.

Questo perchè degli indirizzi indicati dalle quattro etichette vengono considerati solamente i 7 bit meno significativi, come esplicitato dalla sottostante figura:

RP1:RP0	binario	esadecimale	etichetta
00	00100000	0x20	VarB0
00	10100000	0xA0	VarB1
00	1 00100000	0x120	VarB2
00	1 10100000	0x1A0	VarB3
	7 bit		
BANCO	LOCAZIONE		

Figura 2.3: Effetto dei 7 bit meno significativi dell'indirizzo

Le istruzioni del listato 2.2 a pagina 49 vanno quindi riscritte nel seguente modo se si vogliono selezionare correttamente le locazioni desiderate ed ottenere l'effetto voluto:

Listing 2.3: Esempio di indirizzamento corretto

```
;Seleziona il banco 0 della Data Memory
bcf     STATUS,RP1
bcf     STATUS,RP0

;Indirizzamento corretto di VarB1, VarB2 e VarB3
movlw   0x00            ;Carica la costante 0 nel registro W
movwf   VarB0           ;Azzera correttamente VarB0
bsf     STATUS,RP0      ;Seleziona il banco 1
movwf   VarB1           ;Azzera correttamente VarB1
bcf     STATUS,RP0      ;Seleziona il banco 2
bsf     STATUS,RP1
movwf   VarB2           ;Azzera correttamente VarB2
bsf     STATUS,RP0      ;Seleziona il banco 3
movwf   VarB3           ;Azzera correttamente VarB3
```

E' piuttosto evidente che la suddivisione in banchi crea qualche problema al programmatore. E' piuttosto facile fare confusione ed indirizzare in maniera errata le variabili. Al fine di minimizzare tale possibilità si consigliano due accorgimenti:

- raggruppare funzionalmente le variabili a seconda della destinazione d'uso;
- avere a portata di mano la mappatura delle varibili quando si scrive il codice.

In particolare, il primo accorgimento necessita di qualche spiegazione suppletiva.

Si supponga che un determinato processo richieda l'acquisizione di un valore mediante il convertitore analogico-digitale; che tale valore debba essere convertito in decimale; che debba, infine, essere sommato ad un risultato parziale.

Si avrà quindi bisogno di una variabile per l'acquisizione del dato, di una variabile per la conversione ed una per la somma. Con ogni probabilità una mappatura funzionale prevede che le tre variabili siano messe tutte nello stesso banco, al fine di non dover continuamente cambiare banco di memoria durante l'elaborazione del processo.

Una tale modalità di mappatura è sicuramente utile nella maggioranza dei casi.

Il linguaggio di programmazione fornisce, infine, un piccolo aiuto nello *switch* del banco. Si tratta di un'istruzione (in realtà si tratta di una macro, ovvero di una serie di istruzioni raggruppate in una sola) che facilità l'individuazione e la selezione del banco: si tratta dell'istruzione `banksel`. Essa va usata nel seguente modo:

Listing 2.4: Esempio d'uso di banksel

```
movlw   0x00    ;Carica la costante 0 nel registro W
banksel VarB0   ;Seleziona il banco ove e' allocata VarB0
movwf   VarB0   ;Azzera correttamente VarB0
banksel VarB1   ;Seleziona il banco ove e' allocata VarB1
movwf   VarB1   ;Azzera correttamente VarB1
banksel VarB2   ;Seleziona il banco ove e' allocata VarB2
movwf   VarB2   ;Azzera correttamente VarB2
banksel VarB3   ;Seleziona il banco ove e' allocata VarB3
movwf   VarB3   ;Azzera correttamente VarB3
```

La macro `banksel` identifica quindi automaticamente il banco ove una determinata variabile è allocata, senza bisogno che il programmatore sappia in quale banco risiede ciascuna variabile. La mappatura delle variabili rimane comunque un buon suggerimento per non dover richiamare la macro `banksel` un numero improprio di volte.

Lo studente attento avrà sicuramente notato nella mappatura di figura 2.1 a pagina 46 che alcuni registri ad uso specifico come, ad esempio, lo STATUS, il FSR, il PCLATH, l'INTCON, ecc. sono replicati in tutti e quattro i banchi. Ciò

significa che il registro STATUS, ad esempio, è raggiungibile da uno qualsiasi dei quattro banchi.

Ciò è anche ovvio, dato che lo STATUS è proprio il registro che contiene i bit di selezione del banco. E' meno ovvio per gli altri registri. Essi sono, però, registri di una certa importanza e risulta estremamente utile (se non necessario, talvolta) che essi siano disponibili in ciascun banco.

Analogo discorso vale per i registri ad uso generale. In fig. 2.1 a pagina 46, nei banchi 1, 2 e 3 appaiono delle aree di memoria definite **Accesses 70h-7Fh**. Ciò significa che una variabile allocata in una di tali aree di memoria è "raggiungibile" da uno qualsiasi dei quattro banchi.

Le suindicate aree di memoria vanno usate per variabili ad uso generale ritenute dal programmatore particolarmente importanti e comunque tali da essere indirizzabili sempre senza alcuna selezione di banco.

2.1.3 I banchi nell'indirizzamento indiretto

La selezione della locazione di memoria, nell'indirizzamento indiretto, avviene come rappresentato nella figura 2.4.

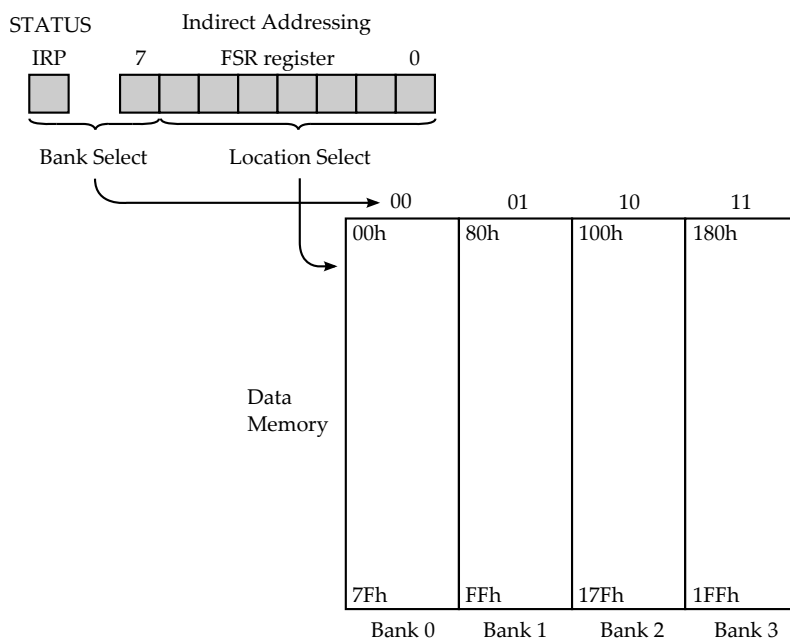



Figura 2.4: Indirizzamento indiretto

Come si evince dalla figura la selezione della locazione di memoria nel banco e la selezione del banco sono leggermente diverse rispetto a quanto avviene nell'indirizzamento diretto.

La selezione della locazione di memoria all'interno del banco selezionato avviene, analogamente al caso precedente, mediante i 7 bit meno significativi del registro FSR.

La selezione del banco, invece, non avviene mediante due bit posti nel registro STATUS, bensì mediante un solo bit posto nel suddetto registro e mediante il bit più significativo del registro FSR.

Nel listato seguente si ripropone l'azzeramento di un vettore già illustrato nel listato 1.4 a pagina 15, ma effettuato quattro volte, una per banco.

Listing 2.5: Esempio di indirizzamento indiretto

```
        ;Seleziona il banco 0
        bcf     STATUS, IRP
        movlw   0x20    ;Inizializza il puntatore FSR

        movwf   FSR
Ciclo0  clrf    INDF    ;Azzera locazione puntata da FSR
        incf    FSR     ;Aggiorna il puntatore
        btfss   FSR, 4  ;Fine azzeramento?
        goto    Ciclo0

        ;Seleziona il banco 1
        movlw   0xA0    ;Inizializza il puntatore FSR

        movwf   FSR
Ciclo1  clrf    INDF    ;Azzera locazione puntata da FSR
        incf    FSR     ;Aggiorna il puntatore
        btfss   FSR, 4  ;Fine azzeramento?
        goto    Ciclo1

        ;Seleziona il banco 2
        bsf     STATUS, IRP
        movlw   0x20    ;Inizializza il puntatore FSR

        movwf   FSR
Ciclo2  clrf    INDF    ;Azzera locazione puntata da FSR
        incf    FSR     ;Aggiorna il puntatore
        btfss   FSR, 4  ;Fine azzeramento?
        goto    Ciclo2

        ;Seleziona il banco 3
        movlw   0xA0    ;Inizializza il puntatore FSR

        movwf   FSR
Ciclo3  clrf    INDF    ;Azzera locazione puntata da FSR
        incf    FSR     ;Aggiorna il puntatore
        btfss   FSR, 4  ;Fine azzeramento?
        goto    Ciclo3
        ;ecc. ecc.
```

In tal modo si azzerano quattro vettori da 16 byte posti rispettivamente agli indirizzi 0x20, 0xA0, 0x120 e 0x1A0.

I consigli forniti nel precedente sottoparagrafo rimangono validi: conviene raggruppare efficacemente e funzionalmente le variabili in modo da evitare frequenti *switch* di banco, non tanto per il rallentamento dovuto all'istruzione, quanto perché la mancata o errata selezione del corretto banco di memoria è fonte di subdoli *bugs* (errori di programmazione) sempre molto difficili da trovare.

Anche nel caso dell'indirizzamento indiretto il linguaggio assembly fornisce un piccolo aiuto nello *switch* del banco. Si tratta della macro `bankisel`. Essa va usata nel seguente modo:

Listing 2.6: Esempio d'uso di bankisel

```
VarB0   equ     0x20    ;Allocazione variabile in banco 0
VarB1   equ     0xA0    ;Allocazione variabile in banco 1
VarB2   equ     0x120   ;Allocazione variabile in banco 2
VarB3   equ     0x1A0   ;Allocazione variabile in banco 3
        ...

        ;Seleziona il banco 0
        bankisel        VarB0
        movlw   VarB0   ;Inizializza il puntatore FSR

        movwf   FSR
Ciclo0  clrf    INDF    ;Azzera locazione puntata da FSR
        incf    FSR     ;Aggiorna il puntatore
        btfss   FSR, 4  ;Fine azzeramento?
        goto    Ciclo0

        ;Seleziona il banco 1
        bankisel        VarB1
        movlw   VarB1   ;Inizializza il puntatore FSR

        movwf   FSR
Ciclo1  clrf    INDF    ;Azzera locazione puntata da FSR
        incf    FSR     ;Aggiorna il puntatore
        btfss   FSR, 4  ;Fine azzeramento?
        goto    Ciclo1

        ;Seleziona il banco 2
        bankisel        VarB2
        movlw   VarB2   ;Inizializza il puntatore FSR

        movwf   FSR
Ciclo2  clrf    INDF    ;Azzera locazione puntata da FSR
        incf    FSR     ;Aggiorna il puntatore
        btfss   FSR, 4  ;Fine azzeramento?
        goto    Ciclo2

        ;Seleziona il banco 3
        bankisel        VarB3
        movlw   VarB3   ;Inizializza il puntatore FSR

        movwf   FSR
Ciclo3  clrf    INDF    ;Azzera locazione puntata da FSR
        incf    FSR     ;Aggiorna il puntatore
        btfss   FSR, 4  ;Fine azzeramento?
        goto    Ciclo3
        ;ecc. ecc.
```

La macro `bankisel` funziona come la `banksel`, solo che la prima è usata nell'indirizzamento indiretto, la seconda nell'indirizzamento diretto.

2.2 La *FLASH Program Memory*

La memoria di programma è rappresentata dalla *FLASH Program Memory*. Il primo termine indica la tecnologia di costruzione, mentre il secondo la destinazione d'uso.

La tecnologia *Flash* succede alla *OTP* (*One Time Programming*) degli anni novanta. Rispetto alla *OTP* la tecnologia *Flash* possiede almeno due evidenti vantaggi, come si deduce dalla sottostante tabella:

Memoria	Scrittura	Velocità
OTP	1	Bassa
Flash	$> 10^2$	Alta

Tabella 2.2: Confronto fra memorie OTP e Flash

La tecnologia *OTP*, come dice il nome, permette una sola programmazione della memoria, mentre la *Flash* permette diverse centinaia di riscritture. Tale caratteristica permette, ad esempio, l'upgrade del software nei dispositivi in manutenzione.

Inoltre, la velocità di scrittura nelle *Flash* è maggiore rispetto alla tecnologia precedente.

Anche la *Program Memory* soffre della suddivisione in diverse aree, come la *Data Memory*. Le singole aree non si chiamano, però, banchi (per evitare confusioni), ma **pagine**. L'organizzazione in pagine della memoria di programma è evidenziata in fig. 2.5.

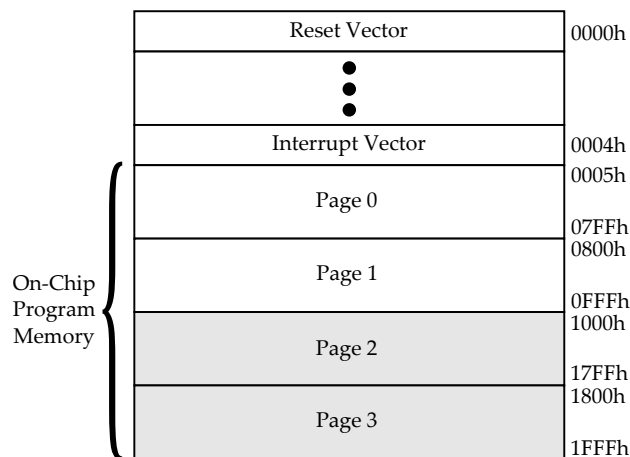



Figura 2.5: Program Memory Map

I PIC16F877 e PIC16F876 possiedono 4 pagine di memoria di programma, per un totale di 8K [1], mentre i PIC16F873 e PIC16F874 possiedono solamente 2

[1] In realtà si dovrebbe scrivere 8KW, indicando in tal modo 8192 *parole* da 14 bit l'una. Frequentemente si commette però l'errore di equiparare il termine *Word* con parole di 16 bit, il che non è sempre vero. Si è pertanto deciso di omettere l'unità di misura, come frequentemente si usa fare nel linguaggio tecnico.

pagine di detta memoria.

Con riferimento alla figura 1.4 a pagina 10 si evince che il *Program Counter* è un registro a 13 bit, ma studiando con un po' di attenzione la fig. 2.1 a pagina 46 si nota che detto registro, in realtà, è suddiviso in due distinti registri: il **PCL** a 8 bit, posto all'indirizzo 02h e il **PCLATH** a 5 bit, posto all'indirizzo 0Ah.

Questi due registri forniscono l'indirizzo completo al *Program Counter*. La modalità con la quale detta operazione avviene è, però, diversa a seconda del contesto: semplice lettura nella *Program Memory* di un'istruzione (indirizzamento diretto) oppure esecuzione di un'istruzione che preveda il registro PCL come destinazione (indirizzameno relativo).

In fig. 2.6 è visualizzato il meccanismo relativo alla lettura dell'istruzione.

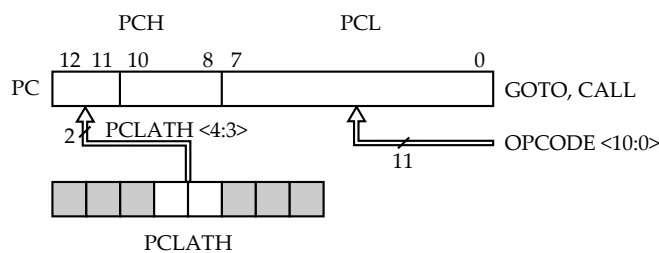



Figura 2.6: Funzione del PCL e del PCLATH

Se l'istruzione letta contiene uno spiazzamento, come la `goto` oppure la `call`, allora gli undici bit meno significativi dell'indirizzo destinazione vengono letti direttamente dall'istruzione (*Opcode*). Detti bit confluiscono in parte nel PCL (8 bit) e parte nella parte bassa del PCH (3 bit), ovvero nella parte alta del *Program Counter*. Si noti che tale parte del registro non compare fra gli *Special Function Registers*.

I restanti due bit necessari a formare l'indirizzo da 13 bit vengono forniti dal registro PCLATH, attraverso i bit 4 e 3. Questi ultimi due bit sono esattamente i bit di selezione della pagina della *Program Memory*.

Se, invece, si sta eseguendo un'istruzione che prevede come destinazione il registro PCL, si deve far riferimento alla figura 2.7.

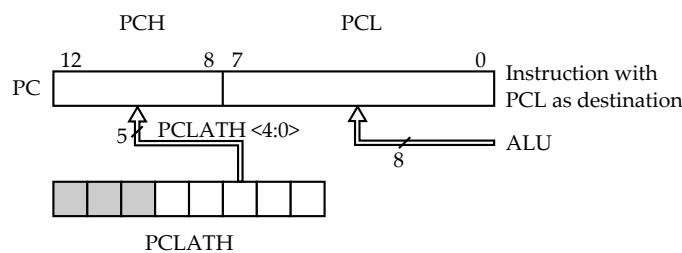



Figura 2.7: Esecuzione di istruzione con PCL come destinazione

Questo secondo caso, che verrà analizzato nel dettaglio nei prossimi paragrafi, è tipico dell'indirizzamento relativo. Quando il registro PCL è destinazione di un'operazione aritmetico-logica l'ALU fornisce gli otto bit meno significativi al PCL, mentre i restanti 5 bit vengono forniti dal PCLATH.

2.2.1 Salti di pagina

Si è visto che i salti di pagina devono essere adeguatamente preparati prima di essere eseguiti, dato che le istruzioni di `goto` e `call` forniscono solamente unidici bit al *Program Counter*. Un esempio di chiamata a subroutine posta in altra pagina da quella chiamante è illustrato nel seguente codice:

Listing 2.7: Esempio di salto di pagina corretto

```
        org     0x500           ;Orgine del codice in 0x500
        bcf     PCLATH, 4       ;Seleziona la pagina 1
        bsf     PCLATH, 3
        call    SubPag1         ;Chiama la routine in pagina 1
        bcf     PCLATH, 3       ;Seleziona la pagina 0
        ...

        org     0x900           ;Origine del codice in 0x900
SubPag1 ...                     ;Esecuzione della SubPag1
        ...
        return                  ;Torna nella pagina chiamante
```

Si noti come lo *Stack* fornisca al *Program Counter* l'indirizzo esteso su 13 bit, eseguendo correttamente il salto all'indirizzo di ritorno, senza alcun aiuto esterno. Si deve, però, porre attenzione che il PCLATH **rimane invariato**, per cui eventuali istruzioni `goto` oppure `call` verranno nuovamente eseguite in pagina 1 se non si provvederà a modificare tale registro.

Si fa anche notare che la modifica del PCLATH non sortisce alcun effetto finché non viene eseguita un'istruzione di salto.

2.3 Esercizi

Gli esercizi riportati nelle seguenti pagine sono tutti relativi a quanto esposto nel capitolo 2. Le soluzioni degli esercizi sono riportate in Appendice A.

Le memorie del PIC16F877

1. ◊◊◊ Qual è la funzione della *Program Memory*?
2. ◊◊◊ Come si scrive la *Program Memory*?
3. ◊◊◊ Qual è la funzione della *Data Memory* (*RAM File Registers*)?
4. ◊◊◊ In quanti banchi è suddivisa la*Data Memory* del PIC16F877?
5. ◊◊◊ Di quanti byte è formato un banco?
6. ◊◊◊ Qual è la particolarità delle aree di memoria 0xF0-0xFF, 0x170-0x17F e 0x1F0-0x1FF?
7. ◊◊◊ Si elenchino almeno 3 registri raggiungibili da qualsiasi banco, senza necessità di selezione.
8. ◊◊◊ Qual è il principale svantaggio della suddivisione in banchi della memoria?
9. ◊◊◊ Qual è il registro coinvolto nella selezione dei banchi?
10. ◊◊◊ Quali sono i bit del suddetto registro coinvolti nella selezione dei banchi nell'indirizzamento diretto?
11. ◊◊◊ Si descriva il meccanismo di accesso alla memoria nell'indirizzamento diretto.
12. ◊◊◊ Quali sono i bit del suddetto registro coinvolti nella selezione dei banchi nell'indirizzamento indiretto?
13. ◊◊◊ Si descriva il meccanismo di accesso alla memoria nell'indirizzamento indiretto.
14. ◊◊◊ Si descriva la principale differenza fra le due macro `banksel` e `bankisel`.
15. ◊◊◊ Quali sono i bit del suddetto registro coinvolti nella selezione dei banchi nell'indirizzamento diretto?
16. ◊◊◊ Si indichino le principali differenze fra le memorie *Flash* e le *OTP*.
17. ◊◊◊ Quali sono gli Special Function Registers coinvolti nell'accesso alla memoria di programma?
18. ◊◊◊ Si descriva il meccanismo di accesso alla memoria di programma nell'indirizzamento diretto.
19. ◊◊◊ Si descriva il meccanismo di accesso alla memoria di programma nell'indirizzamento relativo.
20. ◊◊◊ Si illustri un esempio di salto pagina corretto.

Capitolo 3

I registri del PIC16F877

Nel capitolo precedente si è appreso quale sia l'organizzazione della memoria dati del microcontrollore PIC16F877. Particolarmente imponente può essere apparsa allo studente la fig. 2.1 a pagina 46. Essa raggruppa sostanzialmente tre tipi di registri:

- registri propri del *core* del microcontrollore;
- registri propri della singola periferica;
- registri ad uso generale.

I primi due tipi di registri sono definiti *Special Function Registers*, perché svolgono una funzione ben precisa e particolare e non sono adibiti ad uso generale. Il terzo tipo di registro è di tipo generale (*General Purposes*) e può essere usato liberamente dal programmatore. E' l'area dove il programmatore definisce le proprie variabili.

Nel presente capitolo verranno esaminati i SFR (*Special Function Registers*) del primo tipo, ovvero quelli relativi al *core* del microcontrollore ovvero a quella parte del micro che non comprende le periferiche, ma la parte più centrale formata dall'ALU, dalle memorie e dalla logica di controllo e decodifica.

3.1 Lo *Status Register*

Lo *Status Register* è sicuramente uno dei registri più importanti in qualsiasi microprocessore o microcontrollore. Guardando con attenzione la fig. 2.1 lo si individua subito per almeno due motivi: è posto nei primi indirizzi (all'indirizzo 0x03, per la precisione) ed è replicato in tutti e quattro i banchi: all'indirizzo 0x03 nel banco 0, all'indirizzo 0x83 nel banco 1, all'indirizzo 0x103 nel banco 2 e all'indirizzo 0x183 nel banco 3. Ci si può legittimamente chiedere perché.

Le ragioni fondamentali verranno comprese esaustivamente nel capitolo **??**. Per ora è sufficiente intuire che è particolarmente utile, data la sua importanza, poter accedere al registro di stato in qualunque banco ci si trovi e senza dover preventivamente effettuare un *bank switch*.

L'importanza del registro di stato deriva dai tre particolari compiti che è chiamato a svolgere:

1. mantiene memoria dei *flag* di Carry, Digit Carry e Zero modificati dalle operazioni aritmetico/logiche. Tali *flag* sono di fondamentale importanza per i calcoli su *n* cifre e per i calcoli in virgola mobile;
2. mantiene memoria dei *flag* di $\overline{TO}$ e $\overline{PD}$ che permettono di distinguere le situazioni di *power up* (accensione), intervento del WDT (allarme) e *wake up* (risveglio) dalla modalità di *stand by*;
3. mantiene memoria del banco di memoria attivo sia nell'indirizzamento diretto che indiretto.

La struttura dello *Status Register* è la seguente:

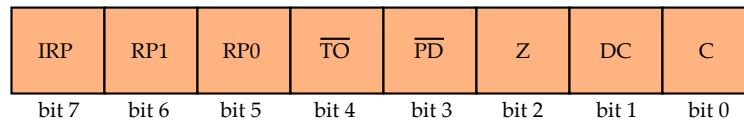



Figura 3.1: Lo *Status Register*

- IRP: Register Bank Select bit (used for indirect addressing)
  1 = Bank 2, 3 (100h - 1FFh)
  0 = Bank 0, 1 (00h - FFh)
- RP1:RP0: Register Bank Select bits (used for direct addressing)
  11 = Bank 3 (180h - 1FFh)
  10 = Bank 2 (100h - 17Fh)
  01 = Bank 1 (80h - FFh)
  00 = Bank 0 (00h - 7Fh)
  Each bank is 128 bytes
- $\overline{TO}$: Time-out bit
  1 = After power-up, clrwdt instruction, or sleep instruction
  0 = A WDT time-out occurred
- $\overline{PD}$: Power-down bit
  1 = After power-up or by the clrwdt instruction
  0 = By execution of the sleep instruction
- Z: Zero bit
  1 = The result of an arithmetic or logic operation is zero
  0 = The result of an arithmetic or logic operation is not zero
- DC: Digit carry/$\overline{borrow}$ bit (addlw, addwf, sublw, subwf instructions)
  (for $\overline{borrow}$, the polarity is reversed)
  1 = A carry-out from the 4th low order bit of the result occurred
  0 = No carry-out from the 4th low order bit of the result
- C: Carry/$\overline{borrow}$ bit (addlw, addwf, sublw, subwf instructions)
  1 = A carry-out from the Most Significant bit of the result occurred
  0 = No carry-out from the Most Significant bit of the result occurred

> Note: For $\overline{borrow}$, the polarity is reversed. A subtraction is executed by adding the two's complement of the second operand. For rotate (rrf, rlf) instructions, this bit is loaded with either the high, or low order bit of the source register.

Il bit di Carry è utilizzato nelle operazioni aritmetiche (addlw, addwf, sublw, subwf) e nelle operazioni di rotazione (rrf, rlf). Nelle operazioni di addizione ha significato di riporto ed è attivo alto. Nelle operazioni di sottrazione ha significato di prestito ed è attivo basso. Nelle operazione di rotazione assume semplicemente il significato di nono bit nella rotazione.

Per una comprensione completa del bit di Carry si vedano le sezioni **??**, **??** e 5.1.

Il bit di Digit Carry assume lo stesso significato del bit di Carry, solamente che esegue il riporto/prestito su 4 bit anziché su 8. Il suo utilizzo è limitato alla somma e sottrazione BCD, le cui cifre, appunto, sono formate da 4 bit anziché 8.

Il bit di Zero assume il valore 1 quando il risultato di un'operazione aritmetico/logica, di incremento o decremento o di spostamento da *file register* vale zero. Si veda a tal proposito la sezione 5.1.

Il bit di $\overline{PD}$, insieme al bit di $\overline{TO}$, riveste un'importanza fondamentale nell'economia del sistema. Esso permette di distinguere un funzionamento "normale" da uno indicante una possibile perdita di controllo del sistema. La tabella 5.1 a pagina 160 illustra come interpretare lo stato del micro in funzione dei bit $\overline{PD}$ e $\overline{TO}$.

All'accensione o in seguito ad un'istruzione di `clrwdt` il bit viene disattivato, ossia posto a 1. Questa situazione è considerata di normalità.

Il bit viene attivato (posto a zero) in seguito ad un'istruzione di `sleep`. Tale situazione è potenzialmente pericolosa, perché il micro disattiva praticamente tutte le periferiche e va in *stand by*. Uno degli eventi che "risvegliano" il micro è lo scadere del WDT, che però, solitamente, indica un malfunzionamento del sistema. Attraverso il presente bit di stato è possibile distinguere le due situazioni.

Il bit di $\overline{TO}$ viene di solito letto in coppia con quello di $\overline{PD}$ (vedi tabella 5.1 a pagina 160). Esso viene disattivato in seguito all'accensione, l'esecuzione dell'istruzione `clrwdt` o quella di `sleep` e viene attivato (posto a zero) in seguito allo scadere del WDT.

I due bit RP0 e RP1 definiscono il banco della memoria dati attivo. Il principale motivo per cui il registro di stato deve essere raggiungibile da qualsiasi banco: la possibilità di selezionare un banco di memoria è dato proprio dal fatto che il registro che permette ciò sia sempre raggiungibile. Inoltre, vi sono situazioni in cui non è possibile sapere a priori quale sia il banco attivo (ad esempio, in seguito ad un'interruzione) ed è fondamentale raggiungere lo *Status Register* indipendentemente dal banco attivo.

Si noti che RP0 e RP1 definiscono il banco attivo per il *solo indirizzamento diretto*.

Il bit IRP permette di distinguere fra i banchi 0, 1 e 2,3 durante gli accessi in memoria eseguiti mediante l'indirizzamento indiretto. Si veda la sezione 2.1.3. Può essere anche utile rileggere il codice 2.5 a pagina 52 per comprendere l'uso del presente bit.

Lo *Status Register*, in virtù della filosofia RISC, pur essendo un registro particolare può essere letto e scritto come qualsiasi altro registro. Ciò necessita alcune attenzioni al fine di evitare pericolosi bachi nel software. E' bene, ad esempio, che la modifica dei bit di stato avvenga solamente mediante le istruzioni `bcf`, `bsf`, `swapf` e `movwf`. Tali istruzioni non modificano il *flag* aritmetici, per cui si è sicuri che uno *switch* di banco non alteri, ad esempio, accidentalmente il bit di Zero o lo stato del Carry. Tale eventualità si presenta durante le operazioni aritmetiche con le variabili coinvolte poste su banchi diversi.

3.2 L'*Option Register*

Un altro registro che occupa uno dei primi indirizzi nella memoria dati è l'*Option Register*[1]. Esso è raggiungibile all'indirizzo 0x81 del banco 1 e all'indirizzo 0x181 del banco 3 e ha il compito di permettere una prima configurazione del micro ed è, di solito, scritto *una tantum* durante linizializzazione del sistema.

L'*Option Register* permette la configurazione delle seguenti parti del micro:

- il *prescaler* del timer TMR0;
- il *postscaler*[2] del WDT;
- l'interrupt esterno facente capo al pin INT;
- il timer TMR0;
- i *pull-ups* del PORTB.

La struttura del registro è la seguente:

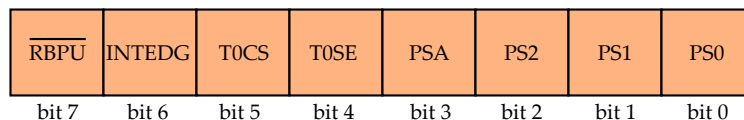



Figura 3.2: L'*Option Register*

- $\overline{RBPU}$: PORTB Pull-up Enable bit
  1 = PORTB pull-ups are disabled
  0 = PORTB pull-ups are enabled by individual port latch values

[1] Si noti che tecnicamente, l'*Option Register* è indicato come OPTION_REG nel file di inclusione fornito dalla Microchip (vedi la sez. 5.3), per cui nel codice esso va indicato in questo secondo modo.

[2] *Prescaler* e *postscaler* sono due termini indicanti un semplice divisore di frequenza posto prima o dopo di un determinato blocco funzionale.

- INTEDG: Interrupt Edge Select bit
  1 = Interrupt on rising edge of RB0/INT pin
  0 = Interrupt on falling edge of RB0/INT pin
- T0CS: TMR0 Clock Source Select bit
  1 = Transition on RA4/T0CKI pin
  0 = Internal instruction cycle clock (CLKOUT)
- T0SE: TMR0 Source Edge Select bit
  1 = Increment on high-to-low transition on RA4/T0CKI pin
  0 = Increment on low-to-high transition on RA4/T0CKI pin
- PSA: Prescaler Assignment bit
  1 = Prescaler is assigned to the WDT
  0 = Prescaler is assigned to the Timer0 module
- PS2:PS0: Prescaler Rate Select bits

PS2:PS0	TMR0 Rate	WDT Rate
000	1:2	1:1
001	1:4	1:2
010	1:8	1:4
011	1:16	1:8
100	1:32	1:16
101	1:64	1:32
110	1:128	1:64
111	1:256	1:128

Tabella 3.1: Prescaler settings

Il bit $\overline{RBPU}$ permette l'abilitazione o meno dei singoli *pull-ups* del PORTB. Ponendo a 0 detto bit si inseriscono sulle linee RB7-RB0 delle resistenze di alto valore collegate verso l'alimentazione positiva del micro. Ciò permette di fissare lo stato logico di dette linee (se programmate come ingressi) debolmente a 1 (*weak pull-up*). La configurazione elettrica è illustrata in fig:3.3 per la sola linea RB0.

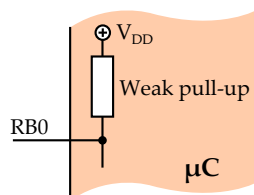



Figura 3.3: *Weak pull-up*

Detta soluzione permette di evitare le ingombranti resistenze di *pull-up* esterne al micro quando si programmano le linee del PORTB come ingressi.

Il bit INTEDG permette la selezione del fronte attivo dell'interruzione esterna. La linea RB0, infatti, è condivisa con la linea INT, che permette, appunto,

l'invio di interruzioni esterne al micro: un cambiamento di fronte richiama automaticamente il vettore di interruzione con relativo salto alla *Interrupt Service Routine*. Il bit INTEDG permette la selezione del suddetto fronte attivo.

Una trattazione esaustiva dell'argomento relativo al PORTB è fornita nella sezione **??**.

I bit T0CS e T0SE permettono rispettivamente la selezione della sorgente del clock del Timer 0 ed il relativo fronte attivo. Se il bit T0CS è posto a 0, viene selezionata la modalità Timer. Ciò significa che il Timer 0 viene incrementato ad ogni ciclo macchina con sorgente del clock interna, ossia CLKOUT. Se il clock fornito al micro provviene da un quarzo a 4MHz, il Timer 0 incrementa ogni μs.

Se il bit T0CS è posto a 1 viene selezionata la modalità Counter. In tal caso il Timer 0 si comporta come contatore con clock esterno sulla linea RA4/T0CKI.

Il bit T0SE permette, invece, la selezione del fronte attivo del clock, nel caso in cui il Timer 0 funzioni in modalità Counter, mentre è ininfluente se è attiva la modalità Timer.

Il bit PSA permette l'assegnazione del *prescaler*. Si tratta di un divisore di frequenza che viene abbinato o al WDT o al Timer 0, a seconda dello stato del presente bit: se il PSA è posto a 1 il *prescaler* viene abbinato al WDT; se è posto a 0 viene abbinato al Timer 0. Se si desidera rendere ininfluente il *prescaler*, si può scegliere un *WDT rate* di 1:1 ed abbinarlo al WDT.

I bit PS2:PS0 permettono il settaggio del *prescaler* secondo la tabella 3.1.

Il *pull-up* interferisce sul corretto funzionamento del *Low Voltage Programming* (vedi sezione 1.10.5). Se si intende utilizzare tale funzione si deve o disabilitare il *pull-up* in tutto il PORTB oppure programmare la linea RB3 (che fa capo al pin di programmazione PGM, utilizzato dal LVP) come uscita in modo da disabilitare il *pull-up* su detta linea. Diversamente la funzione LVP non funziona correttamente.

3.3 Il registro FSR

Il registro FSR è raggiungibile all'indirizzo 0x04 di tutti e quattro i banchi della memoria dati ed è utilizzato per effettuare l'indirizzamento indiretto della memoria. Esso svolge a tutti gli effetti la funzione di puntatore di memoria a 9 bit insieme al bit IRP dello *Status Register*. Siccome $2^9 = 512$ che è esattamente il numero di byte dei quattro banchi di memoria dati, il registro FSR insieme al bit IRP del registro di stato sono sufficienti per indirizzare i quattro banchi di memoria.

Si noti che l'unico indirizzo il cui indirizzamento è privo di significato è l'indirizzo 0x00 di ciascun banco, ossia gli indirizzi 0x00, 0x80, 0x100 e 0x180. Con riferimento alla figura 2.1 a pagina 46 si nota che l'indirizzo 0x00 di ciascun

banco non è un indirizzo fisico vero e proprio. La lettura e la scrittura di tale indirizzo dà quindi luogo a delle eccezioni.

Più precisamente:

- la lettura di detto indirizzo, indipendentemente dal banco selezionato, dà come risultato sempre 0x00;
- la scrittura di detto indirizzo si concretizza con l'esecuzione di una `nop`, ma con l'*effettiva eventuale modifica dei flag di Status*.

A mo' d'esempio si veda il seguente codice:

Listing 3.1: Scrittura dell'indirizzo 0x00

```
1.      movlw   0x00
2.      movwf   FSR     ;Azzera FSR
3.      iorwf   INDF    ;Setta il flag di Zero
```

L'istruzione 3. effettua una scrittura all'indirizzo 0x00, dove non risiede alcun registro fisico, *ma il flag di Zero viene effettivamente settato*.

Si eviti dunque di effettuare azioni di scrittura all'indirizzo 0x00, se non si vogliono alterare i *flag* di Carry, Digit Carry e Zero.

Ulteriori dettagli sul registro FSR si trovano nelle sezioni 1.6.2 e 2.1.3.

3.4 Il registro INTCON

Anche il registro INTCON[3] è raggiungibile da tutti e quattro i banchi di memoria. Esso occupa gli indirizzi 0x0B, 0x8B, 0x10B e 0x18B rispettivamente del banco 0, 1, 2 e 3 e permette una prima gestione delle interruzioni del micro. Più precisamente esso assolve ai seguenti compiti:

- permette l'abilitazione/disabilitazione del *flag* di interruzione generale;
- permette la gestione delle interruzioni legate alle periferiche;
- permette la gestione dell'interruzione legata al Timer 0;
- permette la gestione dell'interruzione esterna;
- permette la gestione dell'interruzione legata al cambiamento di stato delle linee del PORTB.

La struttura del registro è la seguente:

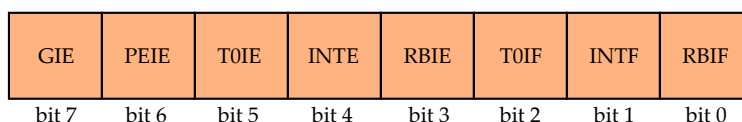



Figura 3.4: Il registro INTCON

[3]INTCON è la contrazione di *INTerrupt CONtrol*.

- GIE: Global Interrupt Enable bit
  1 = Enables all unmasked interrupts
  0 = Disables all interrupts
- PEIE: Peripheral Interrupt Enable bit
  1 = Enables all unmasked peripheral interrupts
  0 = Disables all peripheral interrupts
- T0IE: TMR0 Overflow Interrupt Enable bit
  1 = Enables the TMR0 interrupt
  0 = Disables the TMR0 interrupt
- INTE: RB0/INT External Interrupt Enable bit
  1 = Enables the RB0/INT external interrupt
  0 = Disables the RB0/INT external interrupt
- RBIE: RB Port Change Interrupt Enable bit
  1 = Enables the RB port change interrupt
  0 = Disables the RB port change interrupt
- T0IF: TMR0 Overflow Interrupt Flag bit
  1 = TMR0 register has overflowed (must be cleared in software)
  0 = TMR0 register did not overflow
- INTF: RB0/INT External Interrupt Flag bit
  1 = The RB0/INT external interrupt occurred (must be cleared in software)
  0 = The RB0/INT external interrupt did not occur
- RBIF: RB Port Change Interrupt Flag bit
  1 = At least one of the RB7:RB4 pins changed state; a mismatch condition will continue to set the bit. Reading PORTB will end the mismatch condition and allow the bit to be cleared (must be cleared in software).
  0 = None of the RB7:RB4 pins have changed state

Il bit GIE permette l'abilitazione/disabilitazione globale delle interruzioni. Ponendo tale bit a 0, tutte le interruzioni vengono disabilitate. Se il bit GIE viene posto a 1 vengono abilitate tutte le interruzioni non mascherate, ovvero non già singolarmente disabilitate.

Si noti che tale bit viene automaticamente disabilitato in seguito ad una interruzione, in modo che la ISR non possa essere a sua volta interrotta. Se l'uscita dalla ISR avviene correttamente mediante un'istruzione di `retfie`, quest'ultima riabilita automaticamente le interruzioni settando il bit GIE (vedi sez. 5.1.25).

Il bit PEIE svolge un ruolo del tutto simile: abilita/disabilita le sole interruzioni relative alle periferiche. Ponendo tale bit a 0, tutte le interruzioni relative alle periferiche vengono disabilitate. Se il bit PEIE viene posto a 1 vengono abilitate tutte le interruzioni non mascherate relative alle periferiche.

I bit T0IE e T0IF gestiscono le interruzioni relative al Timer 0. Più precisamente, il bit T0IE, se posto a 1, abilita le interruzioni del Timer e se posto a 0 le disabilita. Il bit T0IF permette di rilevare un'interruzione legata al Timer 0: se il bit è posto a 1 significa che c'è stato *overflow* da parte del Timer 0 ed è stata lanciata la relativa interruzione. In tal caso, è compito del programmatore resettare detto bit affinché possa essere notificata la prossima interruzione. Se, con T0IE = 1, il bit T0IF è posto a 0, significa che non vi è stato *overflow* da parte del Timer 0.

I bit INTE e INTF gestiscono le interruzioni esterne presenti sul pin INT del micro. Più precisamente il bit INTE, se posto a 1, abilita le interruzioni esterne, mentre se posto a 0 le disabilita. Il bit INTF permette di rilevare la presenza di un'interruzione esterna: se il bit è posto a 1 significa che è stata rilevata un'interruzione sulla linea INT e se il bit è posto a 0 significa che non è stata rilevata alcuna interruzione esterna. Se il bit T0IF è posto a 1, è compito del programmatore resettare il bit al fine di permettere la notifica di ulteriori interruzioni.

I bit RBIE e RBIF gestiscono le interruzioni sul cambio di stato delle linee RB7:RB4. Ciò significa che ogni qualvolta, con RBIE=1, una di tali linee, essendo programmata come linea d'ingresso, dovesse cambiare stato logico, verrebbe lanciata un'interruzione al micro. Si noti che cambiamenti di stato di tali linee, se programmate come linee d'uscita, non generano alcuna interruzione.Il bit RBIF, se posto a 1, indica se è stata rilevata interruzione da cambiamento di stato delle linee RB7:RB4. Non vi è però alcuna indicazione di quale linea abbia (oppure quali linee abbiano) cambiato stato logico.

Anche in questo caso, se il bit RBIF è posto a 1 è compito del programmatore resettarlo.

Per una esaustiva esposizione delle tecniche di interruzione si veda il cap. **??**.

3.5 Il registro PIE1

Il registro PIE1 risiede all'indirizzo 0x8C del banco 1. Esso permette l'abilitazione/disabilitazione delle interruzioni relative alle seguenti periferiche:

- il Parallel Slave Port (solo per PIC16F877 e PIC16F874);
- il convertitore analogico digitale;
- l'USART;
- il Synchronous Serial Port;
- il modulo CCP1;
- il Timer 2;
- il Timer 1.

La struttura del registro è la seguente:

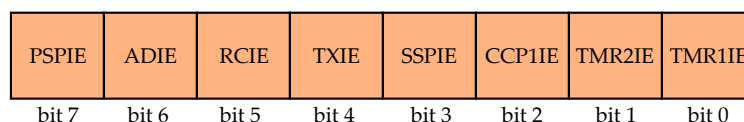



Figura 3.5: Il registro PIE1

- PSPIE[4]: Parallel Slave Port Read/Write Interrupt Enable bit
  1 = Enables the PSP read/write interrupt
  0 = Disables the PSP read/write interrupt

[4]Il bit PSPIE ha significato solamente nei PIC ove il Parallel Slave Port è presente, ovvero nel PIC16F877 e PIC16F874. Nei PIC16F876 e PIC16F873 tale bit è *riservato* e deve essere posto a 0.

- ADIE: A/D Converter Interrupt Enable bit
  1 = Enables the A/D converter interrupt
  0 = Disables the A/D converter interrupt
- RCIE: USART Receive Interrupt Enable bit
  1 = Enables the USART receive interrupt
  0 = Disables the USART receive interrupt
- TXIE: USART Transmit Interrupt Enable bit
  1 = Enables the USART transmit interrupt
  0 = Disables the USART transmit interrupt
- SSPIE: Synchronous Serial Port Interrupt Enable bit
  1 = Enables the SSP interrupt
  0 = Disables the SSP interrupt
- CCP1IE: CCP1 Interrupt Enable bit
  1 = Enables the CCP1 interrupt
  0 = Disables the CCP1 interrupt
- TMR2IE: TMR2 to PR2 Match Interrupt Enable bit
  1 = Enables the TMR2 to PR2 match interrupt
  0 = Disables the TMR2 to PR2 match interrupt
- TMR1IE: TMR1 Overflow Interrupt Enable bit
  1 = Enables the TMR1 overflow interrupt
  0 = Disables the TMR1 overflow interrupt

I singoli bit servono a ad abilitare, ponendo il relativo *flag* a 1, oppure disabilitare, ponendo il *flag* a 0, la rispettiva periferica. Si noti che affinché una periferica risulti avere effettivamente la propria interruzione abilitata è necessaria la concomitanza dei seguenti tre stati:

- relativo bit settato nel registro PIE1 (o PIE2, come si vedrà nella sez. 3.7);
- bit PEIE settato nel registro INTCON;
- bit GIE settato nel registro INTCON.

La latenza (ovvero il tempo fra l'evento che lancia l'interruzione e la notifica attraverso il corrispondente bit di *Interrupt Flag*) è di 3-4 cicli macchina, in dipendenza dell'esatto momento in cui l'evento si produce durante il ciclo macchina corrente.

L'USART ha la capacità di lanciare due distinte interruzioni, sia per carattere trasmesso (TXIE) che per carattere ricevuto (RCIE).

Si noti anche, che se dovesse verificarsi un evento prodotto da una periferica che lancia interruzione, mentre il relativo bit di abilitazione è attivo, ma con il bit PEIE o GIE disattivi, detto evento non va perso e l'interruzione viene notificata non appena i bit PEIE e GIE dovessero essere contemporaneamente attivi.

In tal modo si è sicuri che durante l'asservimento della ISR non vada persa alcuna interruzione lanciata da periferica.

3.6 Il registro PIR1

Il registro PIR1 contiene i bit di notifica delle periferiche elencate nella sezione precedente.

Il registro PIR1 è raggiungibile all'indirizzo 0x0C del banco 0. Quindi per passare dal registro delle abilitazioni (PIE1) al registro delle notifiche (PIR1) è sufficiente modificare il banco, senza modificare l'indirizzo relativo.

Se si produce un'interruzione da parte di qualche periferica, il relativo bit viene posto a 1, altrimenti rimane a 0.

La struttura del registro è la seguente:

Figura 3.6: Il registro PIR1

- PSPIF[5]: Parallel Slave Port Read/Write Interrupt Flag bit
  1 = A read or a write operation has taken place (must be cleared in software)
  0 = No read or write has occurred
- ADIF: A/D Converter Interrupt Flag bit
  1 = An A/D conversion completed
  0 = The A/D conversion is not complete
- RCIF: USART Receive Interrupt Flag bit
  1 = The USART receive buffer is full
  0 = The USART receive buffer is empty
- TXIF: USART Transmit Interrupt Flag bit
  1 = The USART transmit buffer is empty
  0 = The USART transmit buffer is full
- SSPIF: Synchronous Serial Port (SSP) Interrupt Flag
  1 = The SSP interrupt condition has occurred, and must be cleared in software before returning from the Interrupt Service Routine. The conditions that will set this bit are:
  - SPI
    A transmission/reception has taken place.
  - I2C Slave
    A transmission/reception has taken place.
  - I2C Master
    A transmission/reception has taken place.
    The initiated START condition was completed by the SSP module.
    The initiated STOP condition was completed by the SSP module.
    The initiated Restart condition was completed by the SSP module.
    The initiated Acknowledge condition was completed by the SSP module.
    A START condition occurred while the SSP module was idle (Multi-Master system).
    A STOP condition occurred while the SSP module was idle (Multi-Master system).

  0 = No SSP interrupt condition has occurred.

[5] Il bit PSPIE ha significato solamente nei PIC ove il Parallel Slave Port è presente, ovvero nel PIC16F877 e PIC16F874. Nei PIC16F876 e PIC16F873 tale bit è *riservato* e deve essere posto a 0.

- CCP1IF: CCP1 Interrupt Flag bit
  Capture mode:
  1 = A TMR1 register capture occurred (must be cleared in software)
  0 = No TMR1 register capture occurred
  Compare mode:
  1 = A TMR1 register compare match occurred (must be cleared in software)
  0 = No TMR1 register compare match occurred
  PWM mode:
  Unused in this mode
- TMR2IF: TMR2 to PR2 Match Interrupt Flag bit
  1 = TMR2 to PR2 match occurred (must be cleared in software)
  0 = No TMR2 to PR2 match occurred
- TMR1IF: TMR1 Overflow Interrupt Flag bit
  1 = TMR1 register overflowed (must be cleared in software)
  0 = TMR1 register did not overflow

Alcuni bit di notifica, se posti a 1 dal sistema, devono essere resettati dal programmatore (PSPIF, CCP1IF, TMR2IF e TMR1IF), mentre altri no. Ciò dipende dal tipo di periferica che lancia l'interruzione e dal fatto che alcune periferiche necessitano, dopo l'interruzione, l'intervento del programmatore. Tale intervento permette il reset automatico del bit di notifica.

Una corretta comprensione dei sopra elencati bit è possibile solamente dopo aver studiato le relative periferiche, per cui si rimanda una discussione più dettagliata di detti bit al capitolo 4.

3.7 Il registro PIE2

Il registro PIE2 è concettualmente del tutto simile al registro PIE1. E' utilizzato perché un solo registro a 8 bit non è sufficiente a contenere tutti i bit di abilitazione necessari a gestire tutte le periferiche del micro. Il registro è accessibile all'indirizzo 0x8D del banco 1, ovvero l'indirizzo seguente quello del registro PIE1.

Esso contiene le abilitazioni/disabilitazioni di soli 3 bit, come evidenziato dalla figura sottostante:

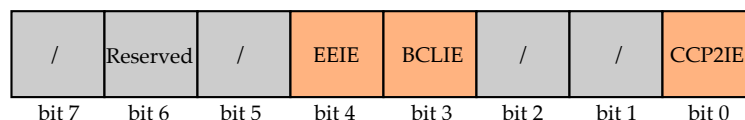



Figura 3.7: Il registro PIE2

- bit 7: Unimplemented. Read as '0'
- bit 6: Reserved. Always maintain this bit clear
- bit 5: Unimplemented. Read as '0'

- EEIE: EEPROM Write Operation Interrupt Enable
  1 = Enable EE Write Interrupt
  0 = Disable EE Write Interrupt
- BCLIE: Bus Collision Interrupt Enable
  1 = Enable Bus Collision Interrupt
  0 = Disable Bus Collision Interrupt
- bit 2-1: Unimplemented. Read as '0'
- CCP2IE: CCP2 Interrupt Enable bit
  1 = Enables the CCP2 interrupt
  0 = Disables the CCP2 interrupt

I bit 7, 5, 2 e 1 non sono implementati e forniscono sempre il valore 0, se letti. Il bit 6 è riservato e deve essere sempre mantenuto a 0.

I restanti bit permettono l'abilitazione/disabilitazione delle interruzioni relative alla scrittura in EEPROM, alla collisione sul bus SSP e al modulo CCP2. Una trattazione più dettagliata verrà fornita nella rispettiva sezione del cap. 4.

3.8 Il registro PIR2

Il registro PIR2 contiene i bit di notifica relativi alle abilitazioni esposte nella sezione precedente.

Il registro PIR2 è raggiungibile all'indirizzo 0x0D del banco 0. Quindi per passare dal registro delle abilitazioni (PIE2) al registro delle notifiche (PIR2) è sufficiente modificare il banco, senza modificare l'indirizzo relativo.

Se si produce un'interruzione da parte di qualche periferica, il relativo bit viene posto a 1, altrimenti rimane a 0.

La struttura del registro è la seguente:

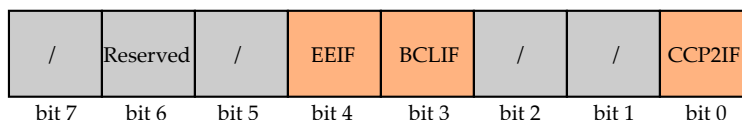



Figura 3.8: Il registro PIR2

- bit 7: Unimplemented. Read as '0'
- bit 6: Reserved. Always maintain this bit clear
- bit 5: Unimplemented. Read as '0'
- EEIF: EEPROM Write Operation Interrupt Flag bit
  1 = The write operation completed (must be cleared in software)
  0 = The write operation is not complete or has not been started
- BCLIF: Bus Collision Interrupt Flag bit
  1 = A bus collision has occurred in the SSP, when configured for I2C Master mode
  0 = No bus collision has occurred
- bit 2-1: Unimplemented. Read as '0'

- CCP2IF: CCP2 Interrupt Flag bit
  Capture mode:
  1 = A TMR1 register capture occurred (must be cleared in software)
  0 = No TMR1 register capture occurred
  Compare mode:
  1 = A TMR1 register compare match occurred (must be cleared in software)
  0 = No TMR1 register compare match occurred
  PWM mode:
  Unused

I bit 7, 5, 2 e 1 non sono implementati e forniscono sempre il valore 0, se letti. Il bit 6 è riservato e deve essere sempre mantenuto a 0.

Alcuni bit di notifica, se posti a 1 dal sistema, devono essere resettati dal programmatore (EEIF, CCP2IF), mentre il bit BCLIF no. La motivazione di tale differenziazione è già stata esposta nella sezione 3.6.

3.9 Il registro PCON

Il registro PCON è raggiungibile all'indirizzo 0x8E del banco 1. Esso permette di distinguere i seguenti quattro eventi:

- un *Power-on Reset* (POR);
- un *Brown-out Reset* (BOR);
- un *Watchdog Reset* (WDT);
- un reset hardware mediante il pin $\overline{MCLR}$.

La struttura del registro è la seguente:

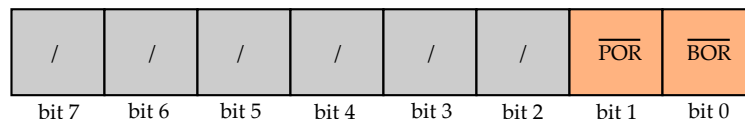



Figura 3.9: Il registro PCON

- bit 7-2: Unimplemented. Read as '0'
- $\overline{POR}$: Power-on Reset Status bit
  1 = No Power-on Reset occurred
  0 = A Power-on Reset occurred (must be set in software after a Power-on Reset occurs)
- $\overline{BOR}$: Brown-out Reset Status bit
  1 = No Brown-out Reset occurred
  0 = A Brown-out Reset occurred (must be set in software after a Brown-out Reset occurs)

Si noti che lo stato del bit $\overline{BOR}$ non è noto all'accensione. Inoltre, se il circuito di *brown-out* non è abilitato mediante l'apposito bit BODEN nel registro di configurazione, il suo stato diventa privo di significato.

Non essendo noto lo stato del bit all'accensione, si deve eseguire una precisa procedura di inizializzazione in seguito ad un POR:

1. valutare il bit $\overline{POR}$ e procedere con i prossimi *step* solo se $\overline{POR} = 0$;
2. settare il bit di $\overline{BOR}$;
3. settare il bit di $\overline{POR}$.

Dopo aver effettuato una corretta inizializzazione è possibile distinguere i diversi eventi che hanno richiamato il vettore di Reset in base allo stato dei bit $\overline{TO}$, $\overline{POR}$ e $\overline{BOR}$, come evidenziato nella sottostante tabella:

$\overline{TO}$	$\overline{POR}$	$\overline{BOR}$	Descrizione evento
1	1	0	*Brown-out Reset*
1	0	1	*Power-on Reset*
0	1	1	*WDT Reset*
1	1	1	Reset hardware attraverso il pin $\overline{MCLR}$

Tabella 3.2: Individuazione tipo di reset

3.10 Il registro di configurazione

Il registro di configurazione non fa parte degli *Special Function Registers* e non è mappato nella memoria dati. Esso è mappato all'indirizzo 0x2007 della memoria di programma e non è ragggiungibile mediante le istruzioni del micro. Ciò significa che esso deve essere scritto *una tantum* in sede di programmazione della memoria di programma. Normalmente ciò avviene attraverso il sistema di sviluppo utilizzato dal programmatore per scrivere e *debuggare* il programma.

La struttura del registro è la seguente:

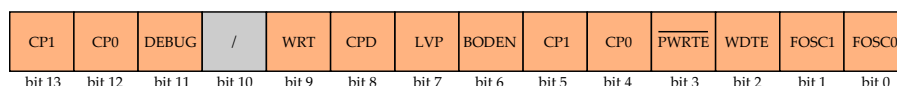



Figura 3.10: Il registro di configurazione

- CP1:CP0: FLASH Program Memory Code Protection bits[6]
  11 = Code protection off
  10 = 1F00h to 1FFFh code protected (PIC16F877, 876)
  10 = 0F00h to 0FFFh code protected (PIC16F874, 873)
  01 = 1000h to 1FFFh code protected (PIC16F877, 876)
  01 = 0800h to 0FFFh code protected (PIC16F874, 873)
  00 = 0000h to 1FFFh code protected (PIC16F877, 876)
  00 = 0000h to 0FFFh code protected (PIC16F874, 873)

[6]Le coppie di bit CP1 e CP0 (bit 13 e 12 e bit 5 e 4) devono avere lo stesso valore per rendere effettiva la protezione del codice secondo lo schema indicato.

- DEBUG: In-Circuit Debugger Mode
  1 = In-Circuit Debugger disabled, RB6 and RB7 are general purpose I/O pins
  0 = In-Circuit Debugger enabled, RB6 and RB7 are dedicated to the debugger.
- bit 10: Unimplemented. Read as '1'
- WRT: FLASH Program Memory Write Enable
  1 = Unprotected program memory may be written to by EECON control
  0 = Unprotected program memory may not be written to by EECON control
- CPD: Data EE Memory Code Protection
  1 = Code protection off
  0 = Data EEPROM memory code protected
- LVP: Low Voltage In-Circuit Serial Programming Enable bit
  1 = RB3/PGM pin has PGM function, low voltage programming enabled
  0 = RB3 is digital I/O, HV on MCLR must be used for programming
- BODEN: Brown-out Reset Enable bit
  1 = BOR enabled
  0 = BOR disabled
- PWRTE: Power-up Timer Enable bit[7]
  1 = PWRT disabled
  0 = PWRT enabled
- WDTE: Watchdog Timer Enable bit
  1 = WDT enabled
  0 = WDT disabled
- FOSC1:FOSC0: Oscillator Selection bits
  11 = RC oscillator
  10 = HS oscillator
  01 = XT oscillator
  00 = LP oscillator

I bit CP1 e CP0 permettono la protezone del codice scritto dal programmatore. La necessità di proteggere il codice residente nel microcontrollore nasce dalla semplicità con cui avvenivano i furti di software negli anni '70 e '80[8].

Se tali bit (in entrambe le coppie) assumono il valore 11, la protezione del codice è disabilitata e un qualsiasi tecnico in possesso di elementari nozioni e strumentazione può scaricare il codice esadecimale memorizzato nel microcontrollore.

Se, invece, essi assumono i valori 01, 10, oppure 00 viene attivata la protezione parziale o totale del codice posto nel microcontrollore. In tal caso non è più possibile leggere il contenuto della memoria di programma né cancellarlo. La protezione avviene secondo lo schema indicato.

Il bit di DEBUG, se posto a 0, abilita l*in-circuit debugging* attraverso i bit RB6 e RB7. In tale modalità i due predetti bit non possono essere utilizzati per uti-

[7]L'abilitazione del bit di BODEN abilita automaticamente anche il bit $\overline{PWRTE}$, indipendentemente dallo stato di quest'ultimo. Il programmatore deve verificare che il *Power-up Timer* sia abilitato ogni qualvolta risulti abilitato il *Brown-out Reset*.

[8]Tali furti hanno facilitato la diffusione dei cosidetti *cloni* IBM e APPLE e sono in relazione (in un caso diretta, nell'altro indiretta) con le disavventure commerciali che le due case produttrici hanno vissuto negli anni '90.

lizzi generali dal programmatore essendo dedicati al sistema per la comunicazione con il sistema di sviluppo. Se il bit DEBUG è posto a 1, il programmatore ha piena disponibilità nell'utilizzo dei bit RB6 e RB7.

Il bit WRT, se posto a 1, permette la scrittura della memoria di programma nelle zone non protette dai bit CP1 e CP0. Se detto bit è posto a 0 non è più possibile scrivere nella memoria di programma, nemmeno nelle zone non protette dai bit CP1 e CP0.

Il bit CPD, se posto a 0, permette anche la protezione dei dati posti nella EEPROM. Solitamente la EEPROM viene utilizzata per memorizzarvi dati di configurazione e parametrizzazione del sistema, per cui può effettivamente risultare utile proteggere tali dati da accidentali scritture o cancellazioni.

Il bit LVP, se posto a 1, permette il *low voltage programming*, utilizzando il pin RB3 in bassa tensione per la programmazione del micro. Se il bit LVR è posto a 0 il bit RB3 è a disposizione del programmatore come linea di I/O, ma il micro deve essere programmato in *high voltage* (12V) sul pin $\overline{MCLR}$.

Il bit BODEN, se posto a 1, abilita il *brown-out reset* nel micro. Tale scelta abilita anche automaticamente il bit $\overline{PWRTE}$, anche se esso è posto a 1 nel registro di configurazione. Si veda la sezione 1.9.5 per maggiori dettagli sul *Brown-out Reset*.

Il bit $\overline{PWRTE}$, se posto a 0, permette l'abilitazione del *Power-up Timer* (vedi la sezione 1.10.1 per dettagli), indipendentemente dallo stato del bit di BODEN.

Il bit WDT, se posto a 1, permette di abilitare il *Watchdog Timer*. Si noti che in tal modo esso non è più disabilitabile, conferendo a detto componente del sistema quella indipendenza che lo rende affidabile.

I bit FOSC1 e FOSC0 permettono di programmare il micro in base alla frequenza e alla tecnologia utilizzate per l'oscillatore che alimenta il clock di sistema. Per la programmazione di detti bit si deve far riferimento alla sezione 1.8 e alla tabella 1.2 a pagina 25.

3.11 Esercizi

Gli esercizi riportati nelle seguenti pagine sono tutti relativi a quanto esposto nel capitolo 3. Le soluzioni degli esercizi sono riportate in Appendice A.

I registri del PIC16F877

1. ◊◊◊ Quali sono i compiti dello *Status Register*?
2. ◊◊◊ Quali sono i *flag* aritmetici?
3. ◊◊◊ Quali sono, rispettivamente, i loro compiti?
4. ◊◊♦ Quali *flag* aritmetici ha senso porre a 0 o a 1 manualmente (ovvero non automaticamente in seguito a operazioni aritmetico-logiche)?
5. ◊◊◊ A cosa serve il bit $\overline{PD}$ del registro di stato?
6. ◊◊◊ A cosa serve il bit $\overline{TO}$ del registro di stato?
7. ◊◊◊ A cosa servono i bit RP1 e RP0 del registro di stato?
8. ◊◊◊ A cosa serve il bit IRP del registro di stato?
9. ◊◊◊ Perché è importante che il registro di stato sia raggiungibile da tutti e quattro i banchi?
10. ◊◊◊ Mediante quali istruzioni è preferibile scrivere il registro di stato?
11. ◊◊◊ Quali sono i compiti dell'*Option Register*?
12. ◊◊◊ In quali banchi si trova *Option Register*?
13. ◊◊◊ Cos'è un prescaler?
14. ◊◊◊ Quali sono i componenti del microcontrollore che possono utilizzare il prescaler?
15. ◊◊◊ Cosa si intende per *pull-up*?
16. ◊◊♦ Si faccia un esempio in cui il *pull-up* è necessario.
17. ◊◊◊ Quali linee del micro possono usufruire del *pull-up* interno?
18. ◊◊◊ A cosa serve il bit INTEDG dell*Option Register*?
19. ◊◊◊ A quale pin del micro fa capo l'interruzione esterna?
20. ◊◊◊ A cosa servono rispettivamente i bit T0CS e T0SE dell'OPTION_REG?
21. ◊◊◊ Quali sono i compiti del registro FSR?
22. ◊◊◊ In quali banchi si trova il registro FSR?
23. ◊◊◊ A cosa serve il registro INTCON?
24. ◊◊◊ In quali banchi è posto il registro INTCON?
25. ◊◊◊ A cosa serve il bit GIE del registro INTCON?

26. ◊◊◊ In quali casi viene disabilitato automaticamente il bit GIE?
27. ◊◊◊ Quale istruzione riabilita automaticamente il bit GIE?
28. ◊◊♦ Perché è bene che durante l'esecuzione della *Interrupt Service Routine* il bit GIE sia disattivo?
29. ◊◊◊ A cosa serve il bit PEIE del registro INTCON?
30. ◊◊◊ A cosa servono i bit T0IE e T0IF del registro INTCON?
31. ◊◊◊ A cosa servono i bit INTE e INTF del registro INTCON?
32. ◊◊◊ A cosa servono i bit RBIE e RBIF del registro INTCON?
33. ◊◊◊ Cos'è il *Port Change Interrupt*?
34. ◊◊◊ Quali linee coinvolge il *Port Change Interrupt*?
35. ◊◊◊ A cosa servono i registri PIE1 e PIE2?
36. ◊◊◊ In quale banco si trovano i registri PIE1 e PIE2?
37. ◊◊◊ Quali sono i moduli coinvolti dai registri PIE1 e PIE2?
38. ◊◊◊ A cosa servono i registri PIR1 e PIR2?
39. ◊◊◊ A cosa servono i registri PIR1 e PIR2?
40. ◊◊◊ Che relazione c'è fra i bit GIE, PEIE e i bit dei registri PIE1 e PIE2?
41. ◊◊◊ A cosa serve il registro PCON?
42. ◊◊◊ Che relazione c'è fra i bit del registro PCON e gli eventi di reset?
43. ◊◊◊ A cosa serve il registro di configurazione?
44. ◊◊◊ Dove è situato il registro di configurazione?
45. ◊◊◊ Come si fa a scrivere il registro di configurazione?
46. ◊◊◊ A cosa servono i bit CP1 e CP0 del registro di configurazione?
47. ◊◊◊ A cosa serve il bit DEBUG del registro di configurazione?
48. ◊◊◊ A cosa serve il bit CPD del registro di configurazione?
49. ◊◊◊ A cosa serve il bit BODEN del registro di configurazione?
50. ◊◊◊ A cosa serve il bit WDT del registro di configurazione?

Capitolo 4

Le periferiche del PIC16F877

Nel corso del presente capitolo verranno esaminate con sufficiente dettaglio le singole periferiche del PIC16F877. Esse sono già state elencate e brevemente descritte nella sezione 1.11. Di seguito si propone uno studio più specifico ed approfondito che dovrebbe coprire in maniera esaustiva ogni singola periferica. Il codice che di volta in volta verrà presentato è scritto in assembly.

4.1 I/O *Ports*

Il PIC16F877 possiede ben 5 *Port*[1] di *Input/Output*, per un totale di 33 linee di I/O, così distribuite:

- **PORTA**: 6 linee;
- **PORTB**: 8 linee;
- **PORTC**: 8 linee;
- **PORTD**: 8 linee;
- **PORTE**: 3 linee;

All'accensione, tutte le linee di I/O sono automaticamente programmate come linee di ingresso digitali o (ove presenti) analogiche. Tali linee possono, però, essere riprogrammate in modo da espletare funzionalità diverse a seconda della necessità e della loro collocazione nei rispettivi Port. Ciò significa che molte linee sono multiplexate e condivise da diverse periferiche.

Siccome ciascun Port ha delle caratteristiche proprie conviene esaminarli distintamente uno ad uno.

[1]Per *Port* si intende un insieme (tipicamente 8) di dispositivi di memorizzazione di uno stato logico.

4.1.1 Il PORTA

Il PORTA è un port bidirezionale a 6 bit, ossia 2 in meno dei Port "canonici". Tutte e 6 le linee del Port svolgono da 2 a 3 possibili funzioni diverse, come evidenziato dalla sottostante tabella.

Name	Bit#	Buffer	Function
RA0/AN0	bit0	TTL	Input/output or analog input
RA1/AN1	bit1	TTL	Input/output or analog input
RA2/AN2	bit2	TTL	Input/output or analog input
RA3/AN3/VREF	bit3	TTL	Input/output or analog input or VREF
RA4/T0CKI	bit4	ST	Input/output or external clock input for Timer0. Output is open drain type
RA5/AN4/$\overline{SS}$	bit5	TTL	Input/output or analog input or slave select input for synchronous serial port

Tabella 4.1: Tipologia delle linee del PORTA

Dal punto di vista dell'architettura, invece, le linee del PORTA appartengono a due diverse strutture: le linee RA3-RA0 e la linea RA5 hanno un'architettura come quella indicata in fig. 4.1, mentre la linea RA4, come si vedrà tra breve, ha una struttura leggermente diversa.

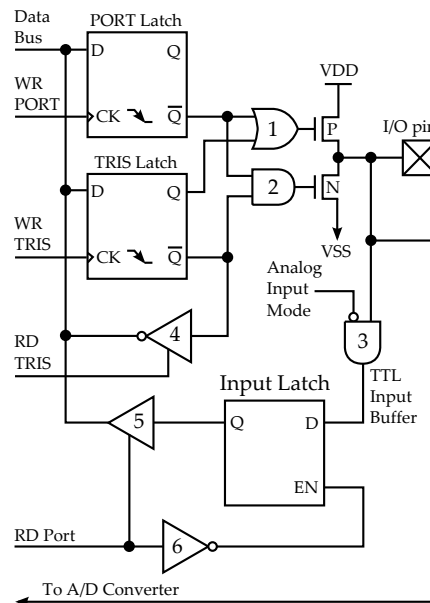



Figura 4.1: Architettura dei pin RA3-RA0 e RA5

Quindi il circuito elettrico sommariamente descritto in fig. 4.1 (relativo ad una sola delle linee RA0, RA1, RA2, RA3 oppure RA5) deve poter svolgere

tutte le funzioni indicate in tabella: deve poter cioè funzionare da linea di ingresso, da linea di uscita o da ingresso analogico[2].

Per semplificare lo studio del predetto circuito lo si ripropone avendo cura di evidenziare in rosso, di volta in volta, le parti relative alle singole funzioni.

4.1.1.1 Il circuito di uscita digitale

In fig. 4.2 è evidenziato sommariamente il circuito di uscita digitale relativo ad una sola delle linee RA0, RA1, RA2, RA3 oppure RA5.

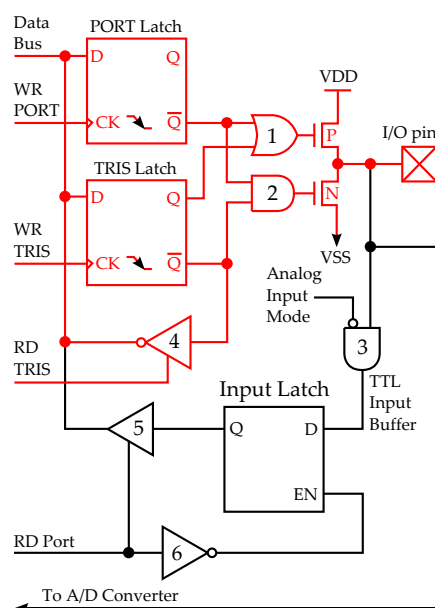



Figura 4.2: Architettura dell'uscita digitale

Il pin è collegato ad un *totem pole* di MOS complementari, pilotati a da una porta OR (MOS a canale P) e da una porta AND (MOS a canale N). Queste, loro volta, sono pilotate dal Latch del PORTA oppure dal Latch del TRISA.

Quando si programma la linea di uscita si deve scrivere uno 0 logico nel Latch del TRISA, che rimane memorizzato finché non si cambia la natura della linea, ad esempio, da uscita ad ingresso.

Il Latch del PORTA viene scritto quando si vuole cambiare lo stato logico di uscita della linea. Quando si vuole portare la linea di uscita alta, si deve scrivere un 1 logico verso il Latch del PORT. Ciò comporta uno uno 0 logico sulla prima linea della porta OR (si ricorda che la seconda linea era già stata posta a 0 in fase di programmazione del Latch del TRIS). Il MOS a canale P viene quindi pilotato con uno 0 logico sul *gate* attivandolo. Per quanto riguarda il MOS a canale N, si ha un 1 logico sulla seconda linea della porta AND (quella pilotata dall'uscita $\overline{Q}$ del Latch del TRISA) ma uno 0 logico sul primo ingresso

[2]La linea RA3 può svolgere anche funzioni di ingresso per la tensione di riferimento dell'ADC (VREF). Una tensione di riferimento è, però, pur sempre una grandezza analogica.

della porta AND (quella pilotata dall'uscita $\overline{Q}$ del Latch del PORTA). In uscita alla porta AND si ha quindi uno 0 logico che interdice il MOS a canale N.

Si ottiene quindi un 1 logico sul pin di I/O.

Quando si scrive, invece, uno 0 sul Latch del PORTA, si porta a 1 il primo ingresso della porta OR, interdicendo il MOS a canale P. Contemporaneamente si porta a 1 anche il primo ingresso della porta AND che assume quindi lo stato logico 1 in uscita, facendo saturare il MOS a canale N e portando a 0 logico il pin di I/O.

Infine, mediante la porta NOT *three state* 4 è possibile leggere lo stato del Latch del TRISA e sapere se la linea è stata programmata come ingresso o come uscita. Il PORTA Latch è leggibile mediante il circuito d'ingresso.

4.1.1.2 Il circuito di ingresso digitale

In fig. 4.3 è evidenziato sommariamente il circuito di ingresso digitale relativo ad una sola delle linee RA0, RA1, RA2, RA3 oppure RA5.

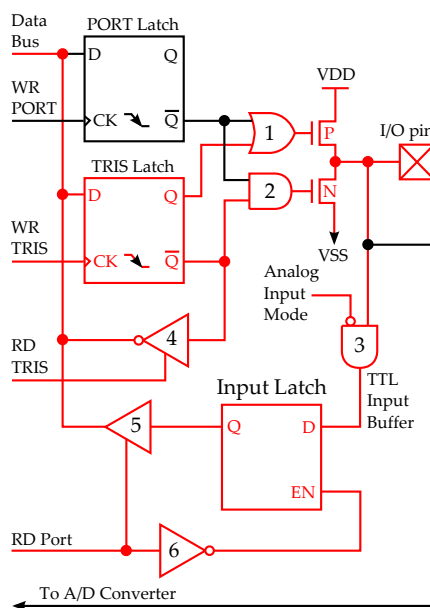



Figura 4.3: Architettura dell'ingresso digitale

Il circuito d'ingresso è un po' più complicato perché la potenziale uscita non deve entrare in conflitto con l'ingresso (ovvero con l'uscita che lo pilota). A tal fine di deve porre in alta impedenza il *totem pole*. Ciò avviene nel seguente modo.

Quando si programma la linea in ingresso, si deve scrivere un 1 logico nel Latch del TRISA. Ciò implica un 1 logico all'uscita della porta OR ed uno 0 logico all'uscita della porta AND. In tal modo sia il MOS a canale P che quello a canale N risultano essere interdetti e non interferiscono in nessun modo con il segnale presente sul pin di I/O.

Se si vuole programmare la linea come ingresso digitale, si deve porre a 0 logico l'ingresso *Analog Input Mode* della porta AND 3. Tale ingresso è attivo basso, per cui lo stato logico del pin di I/O viene replicato all'uscita della porta AND 3. Tale porta funge da *TTL Input Buffer*, il che significa che i livelli statici di ingresso della porta logica sono TTL (ossia più severi dei livelli MOS) e che dal punto di vista dinamico la transazione deve avvenire con i tempi del segnale digitale (qualche nanosecondo).

Quando si vuole leggere il pin di I/O si deve attivare il segnale di lettura *RD Port* il quale "congela"[3] lo stato logico del pin di I/O disabilitando, attraverso la porta NOT 6, il segnale EN dell'*Input Latch* e abilita l'uscita del *buffer three state* 5 in modo da porre sul *Data Bus* lo stato logico congelato del pin di I/O. In tal modo, se durante l'esecuzione dell'istruzione di lettura dovesse variare lo stato logico del pin di I/O, non si avrebbero variazioni di stato sul *Data Bus* e la lettura del pin di ingresso sarebbe sicura e non ambigua.

Mediante la porta NOT *three state* 4 è sempre possibile leggere lo stato del Latch del TRISA e sapere se la linea è stata programmata come ingresso o come uscita.

4.1.1.3 Il circuito di ingresso analogico

In fig. 4.4 è evidenziato sommariamente il circuito di ingresso analogico relativo ad una sola delle linee RA0, RA1, RA2, RA3 oppure RA5, anche se, superficialmente, si potrebbe dire che il circuito d ingresso analogico è formato semplicemente dal filo che collega il pin di I/O all'*A/D Converter*.

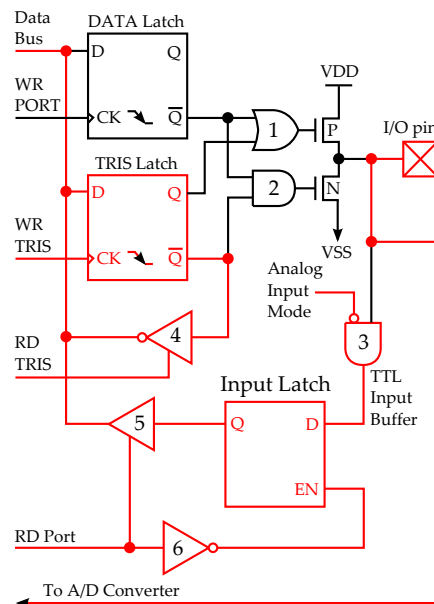



Figura 4.4: Architettura dell'ingresso analogico

[3]Il termine inglese utilizzato nella letteratura tecnica è *to freeze*, da cui "congelare".

Ciò naturalmente non è vero, perché, come per gli ingressi digitali, si deve evitare che il circuito di uscita digitale confligga con quello di ingresso analogico.

Inoltre, le linee che si intendono usare come ingressi analogici *devono comunque essere programmate come ingressi digitali nel TRISA*. Ciò comporta anche la necessità di poter leggere lo stato del Latch TRISA. Con ciò si è spiegato il motivo per cui sono stati evidenziati il TRIS Latch e la porta NOT 4.

Siccome, però, è evidenziata anche la circuiteria che fa capo all'*Input Latch* si deve supporre che anch'esso è coinvolto. Infatti è pure possibile "leggere" lo stato logico dell'ingresso analogico, solo che la lettura restituisce sempre il livello logico 0. Ciò è dovuto dal fatto che l'ingresso *Analog Input Mode* è posto a 1 logico e l'ingresso D dell'*Input Latch* è quindi sempre a 0.

4.1.1.4 La linea RA4

In fig. 4.5 è evidenziato sommariamente il circuito di I/O relativo alla linea RA4.

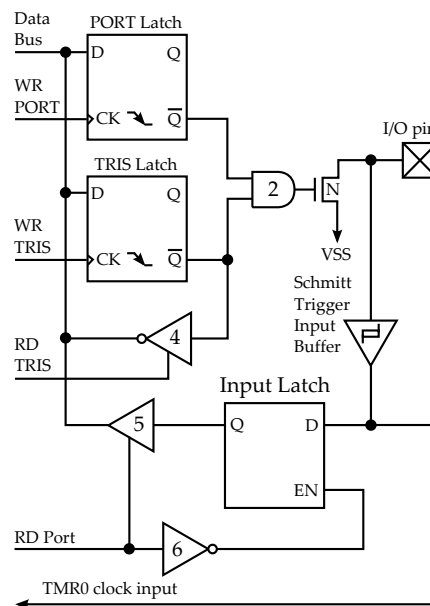



Figura 4.5: Architettura del pin RA4

Esso, pur assomigliando al circuito di fig. 4.1, presenta tre sostanziali differenze:

1. il MOS a canale P, avente funzione di *pull up*, risulta assente;
2. l'*Input Buffer* non è una porta AND TTL, ma una porta NOT triggerata;
3. il pin di I/O è connesso (indirettamente) al clock del Timer 0.

Ciò ha sensibili ripercussioni sul funzionamento del pin RA4, che non può assolutamente essere confuso con i restanti pin del PORTA.

Innanzi tutto se la linea RA4 viene programmata come linea d'uscita, il programmatore deve tener presente che detta uscita è *open drain*, ovvero mancante del componente avente funzioni di *pull up*. E' compito del progettista *aggiungere un resistore di pull up esterno* come evidenziato in fig. 4.6.

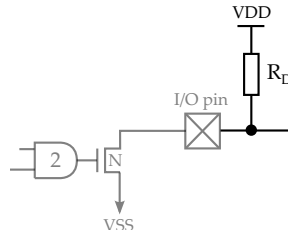



Figura 4.6: Uscita *open drain*

Quando il MOS a canale N è interdetto, lo stato logico 1 deve essere garantito dal resistore di *pull up*, che deve essere adeguatamente dimensionato in relazione al carico visto dalla linea. Quando il MOS è attivo, il resistore di *pull up* funge da resistore di polarizzazione del *drain*.

Il dimensionamento di tale resistore sarà l'oggetto della prossima sezione.

Vi è una differenza sostanziale, rispetto alle restanti linee del PORTA, anche se la linea RA4 viene programmata in ingresso.

Il buffer di ingresso, infatti, non è di tipo TTL, ma di tipo *Schmitt Trigger*. Ciò permette di inviare esternamente al pin di I/O un segnale anche non rigorosamente digitale ma analogico, ovvero lento a piacere. Esso verrà convertito in un segnale digitale dal buffer ST. L'argomento verrà trattato in maggior dettaglio nella sezione 4.1.1.6.

Infine, la linea RA4 presenta un'ultima differenza: essa non può funzionare come linea di ingresso analogico. Può, però, essere utilizzata come clock esterno del Timer 0. In tal caso detta periferica non funziona in modalità timer, ma in modalità contatore e gli eventi da contare sono forniti dall'esterno.

Inoltre, detti eventi possono anche non essere strettamente digitali ma analogici: il buffer Scmitt Trigger renderà digitale qualsiasi segnale fornito al pin di I/O e lo passerà al clock del Timer 0.

4.1.1.5 L'uscita *Open Drain*

Il termine *open drain* indica la mancanza del componente di polarizzazione collegato fra il *drain* del MOS a canale N e l'alimentazione. E' quindi compito del progettista prevedere tale componente e dimensionarlo adeguatamente in relazione al carico.

Avere almeno un'uscita in tecnologia *open drain* può presentare qualche vantaggio.

Permette, ad esempio, l'implementazione di un semplice *bus* di piccole dimensioni, come quelle di una singola scheda elettronica. Potrebbe, a mo' d'esempio, trattarsi di un bus I^2C che, per evitare i conflitti in linea utilizza proprio una tecnologia *open drain*. Si avrà modo, nei prossimi capitoli, di trattare l'argomento.

Il dimensionamento del predetto resistore è sempre da porre in relazione al carico, come appare piuttosto evidente dal circuito di fig. 4.7, dove R_D rappresenta il resistore di *pull up* ed R_L rappresenta il carico generico.

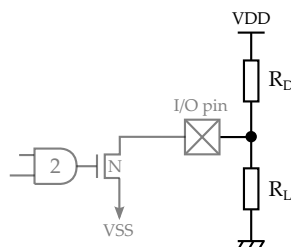



Figura 4.7: Collegamento di un carico all'*open drain*

Frequentemente il carico è null'altro che l'ingresso (o gli ingressi) di una porta logica, che risponde, quindi ad assorbimenti e livelli di tensione chiaramente definiti.

Se il carico è in tecnologia CMOS o MOS, il dimensionamento di R_D è praticamente superfluo, dato che la corrente assorbita in ingresso da un tale dispositivo è sull'ordine del nA.

Se, però, il componente che funge da carico è, ad esempio, in tecnologia TTL, le cose cambiano leggermente.

Una porta TTL riconosce uno stato logico al proprio ingresso quando la tensione ivi applicata è compresa fra 0 e 0.4V. Riconosce, inoltre, uno stato logico alto quando la tensione ivi applicata compresa fra 2 e 5V.

Lo stato logico basso è garantito dalla saturazione del MOS a canale N, mentre lo stato logico alto deve essere garantito dal resistore R_D. Per poterlo dimensionare è necessario conoscere la corrente assorbita dall'ingresso che, nel caso di un *fan-in* = 1, è dichiarata dal costruttore avente valore di $40\mu A$. Supponendo un margine di rumore minimo di 0.4V la relazione che definisce il valore di R_D è la seguente:

$$R_D < \frac{V_{DD} - V_{INH} - V_N}{40 \cdot 10^{-6}} = \frac{2.6}{40 \cdot 10^{-6}} = 65K\Omega \tag{4.1}$$

dove $V_{DD} = 5V$, $V_{INH} = 2V$ e il margine di rumore vale $V_N = 0.4V$. Valori comunemente usati sono $R_D = 47K\Omega$ oppure $R_D = 10K\Omega$.

Se la natura del carico dovesse essere diversa la natura del calcolo non cambierebbe. Nel caso, invece, in cui il carico dovesse essere collegato fra *drain* e alimentazione, il resistore R_D potrebbe essere direttamente omesso.

4.1.1.6 La porta *Schmitt Trigger*

Le porte textitSchmitt Trigger sono normalmente usate in presenza di segnali digitali non molto puliti, oppure quando i fronti di salita e discesa dovessero avere tempi piuttosto consistenti, oppure ancora in presenza di segnali strettamente analogici.

La porta CMOS classica non sopporta, infatti, segnali il cui tempo di transizione dal livello basso a quello alto e viceversa sia troppo alto. Il passaggio da uno stato all'altro deve classicamente avvenire nel'arco temporale massimo di qualche decina di ns. Altrimenti la porta logica potrebbe avere delle difficoltà nel riconoscere correttamente il livello logico d'ingresso e fornire in uscita stati logici non coerenti o addirittura brevi oscillazioni, con conseguenze disastrose per eventuali circuiti sequenziali posti a valle.

La porta *triggerata* (ovvero costruita con tecnologia *Schmitt Trigger*) si comporta differentemente dalla normale porta logica in presenza di segnali dai fronti lenti o addirittura in presenza di segnali analogici, come evidenziato nella figura sottostante.

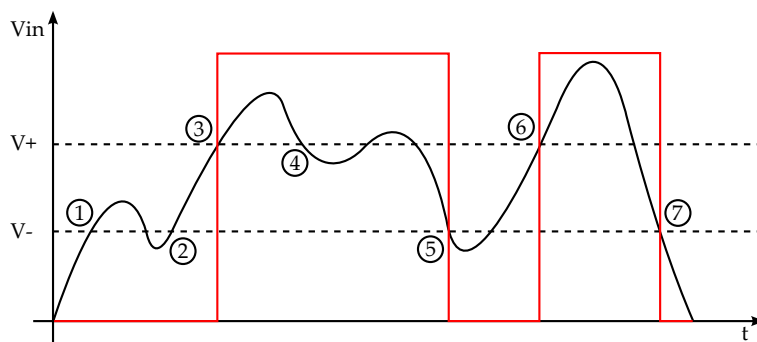



Figura 4.8: Funzionamento buffer *Schmitt Trigger*

La porta triggerata è caratterizzata da due *tensioni di soglia* (*threshold voltages*) chiamate frequentemente V+ (tensione di soglia superiore) e V- (tensione di soglia inferiore). In tecnologia CMOS dette tensioni hanno valore standard pari a $1/3V_{DD}$ (V-) e $2/3V_{DD}$ (V+).

Il grafico di fig.4.8, rappresenta in nero il segnale d'ingresso della porta triggerata ed in rosso la sua Uscita. Detto grafico va interpretato nel seguente modo:

1. se il segnale, analogico e lento a piacere, staziona sotto la tensione di soglia inferiore, l'ingresso lo riconosce come uno stato logico 0 e lo replica in uscita[4]. Se il segnale supera la tensione di soglia inferiore senza superare quella superiore, il segnale d'uscita rimane a 0;
2. se il segnale d'ingresso, essendo stato riconosciuto come stato logico 0, oscilla intorno alla V-, lo stato logico d'uscita non cambia;
3. quando, però, il segnale d'ingresso, già riconosciuto come stato logico 0, supera la tensione di soglia superiore, l'uscita cambia di stato e si porta in stato logico 1;
4. se il segnale d'ingresso, essendo stato riconosciuto come stato logico 1, oscilla intorno alla V+, lo stato logico d'uscita non cambia;
5. per cambiare di stato è necessario che il segnale scenda sotto la tensione di soglia inferiore. In tal modo l'uscita tornerà bassa e l'ingresso sarà riconosciuto come stato logico 0;

[4]Solitamente le porte triggerate sono invertenti, ma nel presente caso il buffer non inverte il segnale d'ingresso.

6. finché il segnale, già riconosciuto come stato logico 0, non supera la tensione di soglia superiore, esso non cambia di stato. Quando il segnale supera la V+ l'uscita si pone a 1;
7. affinché il segnale d'uscita torni a 0, il segnale d'ingresso deve scendere nuovamente sotto la tensione di soglia inferiore.

Si ribadisce ancora che il segnale d'ingresso può avere variazioni lente a piacere. Naturalmente, se il segnale è digitale la porta triggerata lo riconosce correttamente senza difficoltà.

Avere nel pin RA4 un ingresso triggerato è quindi di indubbio vantaggio, potendovi collegare segnali anche non perfetti nei fronti di salita e discesa.

4.1.1.7 Il circuito di uscita digitale della linea RA4

In fig. 4.9 è evidenziato sommariamente il circuito di uscita digitale relativo alla linea RA4.

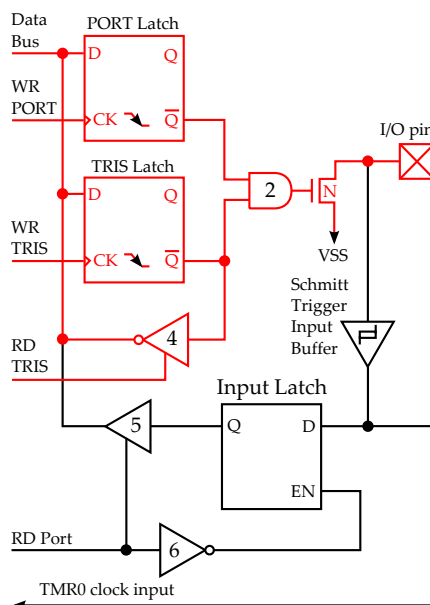



Figura 4.9: Architettura dell'uscita digitale di RA4

Come per le restanti linee del PORTA, anche la linea RA4 può essere programmata come ingresso oppure come uscita mediante il Latch del TRISA e il valore programmato può essere letto attraverso la porta NOT 4. Programmando la linea come uscita, si ha un 1 logico all'uscita $\overline{Q}$ del Latch del TRISA e quindi anche sul secondo ingresso della porta AND.

Scrivendo un 1 logico nel Latch del PORTA, si ha uno 0 logico sul primo ingresso della porta AND che pone, quindi, uno 0 sul *gate* del MOS a canale N che rimane interdetto. L'uscita, in tali condizioni, è flottante.

Scrivendo uno 0 logico nel Latch del PORTA, si ha un 1 logico sul primo ingresso della porta AND che attiva in tal modo il MOS e pone l'uscita a 0 logico.

Mediante la porta NOT *three state* 4 è possibile leggere lo stato del Latch del TRISA e sapere se la linea è stata programmata come ingresso o come uscita. Il PORTA Latch è leggibile mediante il circuito d'ingresso.

4.1.1.8 Il circuito di ingresso digitale della linea RA4

In fig. 4.10 è evidenziato sommariamente il circuito di ingresso digitale relativo alla linea RA4.

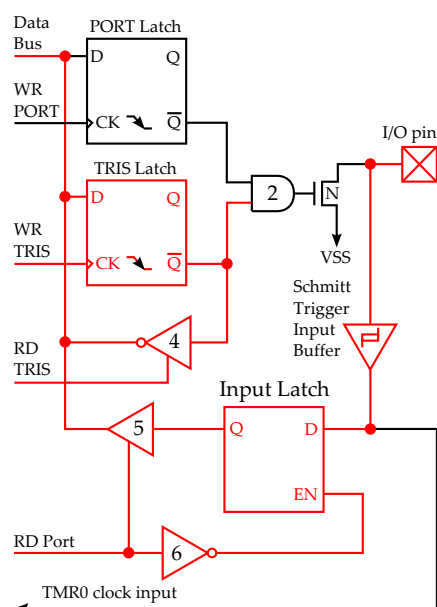



Figura 4.10: Architettura dell'ingresso digitale di RA4

Anche per quanto riguarda il pin RA4 è fondamentale che il circuito di uscita non interferisca con quello di ingresso. E' quindi importante che quando la linea RA4 è programmata in ingresso, il MOS sia interdetto.

Effettivamente è ciò che succede, dato che un 1 logico memorizzato nel Latch del TRISA fa si che ci sia uno 0 logico sul secondo ingresso della porta AND. In tal modo l'uscita di detta porta sarà sicuramente a 0, il che implica l'interdizione del MOS.

Analogamente a quanto visto per i restanti pin del PORTA, è sempre possibile sapere se la linea RA4 è programma in ingresso oppure in uscita, attraverso la lettura del TRISA Latch, che avviene attraverso la porta NOT 4.

La lettura dello stato logico d'ingresso sul pin di I/O, invece, avviene nel seguente modo: attivando il segnale *RD Port* si toglie la trasparenza all'*Input Latch* (che quindi memorizza lo stato logico restituito in quell'istante dal buffer triggerato) e si abilita contemporaneamente il buffer *three state* 5, ponendo il

livello logico (o il corrispondente livello analogico) presente sul pin di I/O sul *Data Bus*.

4.1.1.9 Il circuito di ingresso di clock del Timer 0

In fig. 4.11 è evidenziato sommariamente il circuito di ingresso di clock del Timer 0.

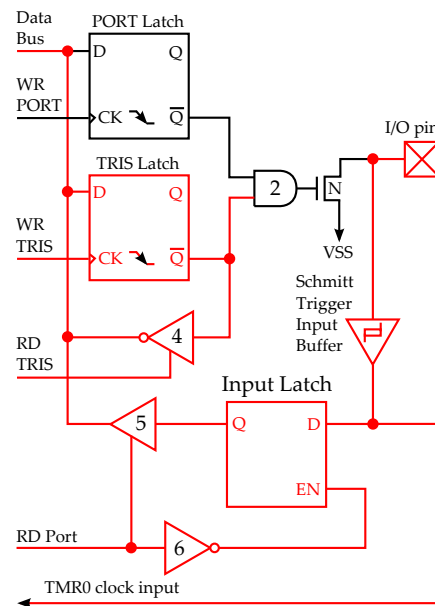



Figura 4.11: Architettura dell'ingresso di clock del Timer 0

Si nota che l'architettura del clock del Timer 0 è del tutto simile a quella dell'ingresso digitale/analogico. L'unica differenza consiste nel fatto che, oltre a essere possibile la lettura del segnale di clock, esso viene direttamente inviato all'ingresso di clock di detto timer.

Utile è il fatto che il segnale che funge da clock al timer passi attraverso il buffer triggerato. In tal modo si aggiunge affidabilità al conteggio, dato che il segnale esterno del clock potrebbe essere affetto da rumore o essere il prodotto di un encoder o altro dispositivo che non restituisce fronti sempre perfetti.

4.1.1.10 La protezione degli ingressi

Anche in virtù di quanto appena detto in chiusura della precedente sezione, si sottolinea il fatto che gli ingressi sono tutti protetti in tensione.

Più precisamente, la linea RA4 è protessa dalle tensioni aventi valore minore di V_{SS} (quindi negative), mentre le restanti linee sono protette sia dalle

tensioni aventi valore minore di V_{SS} che dalle tensioni di valore superiore a V_{DD}.

Lo schema di principio secondo il quale avviene detta protezione è illustrato in fig. 4.12.

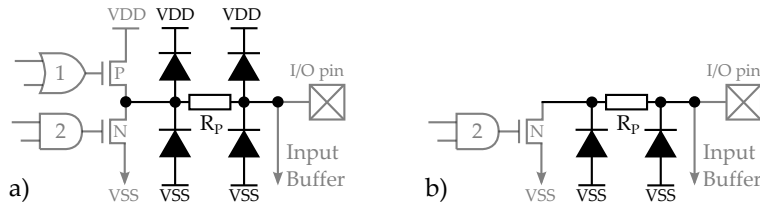



Figura 4.12: Protezione mediante diodi di *clamping*

In fig.4.12a è illustrata la protezione dei pin RA0-RA3, RA5 ed in fig. 4.12b la protezione della linea RA4.

Nel circuito di fig.4.12a una qualsiasi tensione superiore a $V_{DD} + V_\gamma$ viene cortocircuitata verso V_{DD}, dove V_γ rappresenta la tensione di soglia del diodo, che vale circa 0.7V.

Analogamente, in ambedue i circuiti, una tensione minore di $V_{SS} - V_\gamma$ viene cortocircuitata verso V_{SS}.

La protezione è solitamente formata da una doppia serie di diodi, connessi attraverso un resistore del valore di un centinaio di Ohm circa. Essa, naturalmente, non è perfetta, essendo ottenuta mediante diodi parassiti, quindi non in grado di sopportare correnti importanti. Riesce comunque a fornire una buona protezione in ingresso.

4.1.1.11 La programmazione del PORTA

E' compito del programmatore programmare il PORTA come da esigenze hardware, stabilendo, cioè, quali linee debbano essere programmate in ingresso, quali in uscita, quali analogiche, ecc.

La Microchip fornisce un esempio di programmazione del PORTA supponendo che le linee RA0-RA3 debbano essere programmate come ingressi e le linee RA4 e RA5 come uscite[5].

Il codice è il seguente:

Listing 4.1: Inizializzazione PORTA

```
banksel PORTA   ;Select Bank 0
clrf    PORTA   ;Initialize PORTA by clearing output
                ;data latches
banksel TRISA
movlw   0x06    ;Configure all pins
movwf   ADCON1  ;as digital inputs
movlw   0xCF    ;Value used to initialize TRISA
movwf   TRISA   ;Set RA<3:0> as inputs and RA<5:4> as
                ;outputs. TRISA<7:6> are read as 0.
```

[5]Si ricorda che la linea RA4 è comunque un'uscita *open drain*.

Due sono gli aspetti sui quali soffermarsi brevemente.

Prima di programmare il *data direction register* (TRISA) vengono poste le uscite del PORTA ad un valore noto, in modo tale che non vi siano repentine variazioni di stato di dette linee. Se tale inizializzazione venisse fatta dopo aver programmato le linee di uscita, esse potrebbero assumere un valore indeterminato alla prima accensione e subito dopo, per effetto della inizializzazione, cambiare di stato logico.

Un secondo aspetto importante è dato dal fatto che, pur non essendo utilizzato, è necessario programmare parzialmente il convertitore AD, se non altro per disabilitarlo. Ciò avviene scrivendo il registro `ADCON1` con il valore 0x06.

Infine, il datasheet del PIC16F877 riassume, per comodità, quali sono i registri ed i bit coinvolti nella programmazione (solo per quanto riguarda l'I/O digitale) del PORTA. I bit con sfondo grigio non devono essere considerati. In fig. 4.13 è presentata la tabella riassuntiva.

Address	Name	bit 7	bit 6	bit 5	bit 4	bit 3	bit 2	bit 1	bit 0	Value on POR, BOR	Value on all other RESETS
05h, 105h	PORTA	/	/	RA5	RA4	RA3	RA2	RA1	RA0	--0x 0000	--0u 0000
85h, 185h	TRISA	/	/	PORTA Data Direction Register						--11 1111	--11 1111
9Fh	ADCON1	ADFM	/	/	/	PCFG3	PCFG2	PCFG1	PCFG0	--0- 0000	--0- 0000

Legend: x = unknown, u = unchanged, / = unimplemented locations read as 0
Shaded cells are not used by PORTA

Figura 4.13: Registri coinvolti nella programmazione del PORTA

Nella sudetta tabella è evidenziato un particolare che solitamente è piuttosto trascurato, ma non per questo poco importante: lo stato dei bit dei registri indicati dopo un *Power On Reset* (ovvero dopo un'accensione del micro), dopo un *Brown Out Reset* (ovvero dopo una caduta di tensione) e dopo un qualsiasi altro reset. Appare evidente che tutti i bit del registro TRIS si pongono a 1 (ingressi) e che solamente lo stato del pin RA4 non è predicabile dopo una prima accensione o dopo un BOR, mentre tutti gli altri bit si pongono in uno stato noto.

4.1.2 Il PORTB

Il PORTB è un port bidirezionale a 8 bit. Presenta caratteristiche leggermente differenti rispetto al PORTA, che sono essenzialmente riassumibili come segue:

- gestisce la cominicazione con il sistema di sviluppo, attraverso tre linee dedicate;
- permette la programmazione della funzione di *pull up* sulle linee di ingresso;
- permette il lancio di interruzione sul cambiamento di stato delle 4 linee più significative.

Tre sono le linee dedicate al sistema di sviluppo: le linee RB7, RB6 e RB3. Naturalmente, nel caso in cui siano dedicate alla comunicazione con l'emulatore, tali linee non possono essere usate dal sistema. L'argomento è già stato diffusamente trattato nelle sezioni 1.10.4 e 1.10.5, per cui non si ritiene necessario dilungarsi oltre sul tema.

Le 8 linee del port possono far uso di dispositivi di *pull up* sul pin di I/O quando questo è programmato come ingresso. Non si tratta di resistori veri e propri, ma di un MOS a canale P leggermente attivo in modo tale da vincolare debolmente gli ingressi a V_{DD}. Tali *pull up* sono programmabili via software e permettono di risparmiare i resistori esterni.

Inoltre, vi è la possibilità di lanciare un'interruzione al micro ogni qualvolta una delle linee RB7:RB4 cambia il proprio stato logico. Tale particolarità si rivela particolarmente utile nella gestione di pulsanti e tastiere a matrice.

Anche per il PORTB viene proposta una tabella delle funzioni abbinate a ciascuna linea di I/O, come di seguito evidenziato.

Name	Bit#	Buffer	Function
RB0/INT	bit0	TTL/ST	Input/output pin or external interrupt input. Internal software programmable weak pull-up.
RB1	bit1	TTL	Input/output pin. Internal software programmable weak pull-up.
RB2	bit2	TTL	Input/output pin. Internal software programmable weak pull-up.
RB3/PGM	bit3	TTL	Input/output pin or programming pin in LVP mode. Internal software programmable weak pull-up.
RB4	bit4	TTL	Input/output pin (with interrupt-on-change). Internal software programmable weak pull-up.
RB5	bit5	TTL	Input/output pin (with interrupt-on-change). Internal software programmable weak pull-up.
RB6/PGC	bit6	TTL/ST	Input/output pin (with interrupt-on-change) or In-Circuit Debugger pin. Internal software programmable weak pull-up.
RB7/PGD	bit7	TTL/ST	Input/output pin (with interrupt-on-change) or In-Circuit Debugger pin. Internal software programmable weak pull-up.

Tabella 4.2: Tipologia delle linee del PORTB

Come il PORTA, anche il PORTB è formato da due diverse strutture circuitali diverse, che verranno esaminate separatamente.

La prima comprende i 4 bit più significativi del PORTB e la seconda comprende i 4 bit meno significativi.

Dapprima verrà esaminata la struttura relativa ai bit RB7:RB4, che rappresenta la parte più complessa e più versatile del PORTB. Di seguito verrà trattata la parte relativa ai 4 bit meno significativi.

Nella fig. 4.14 è evidenziata l'architettura dei pin di I/O RB7:RB4 del PORTB.

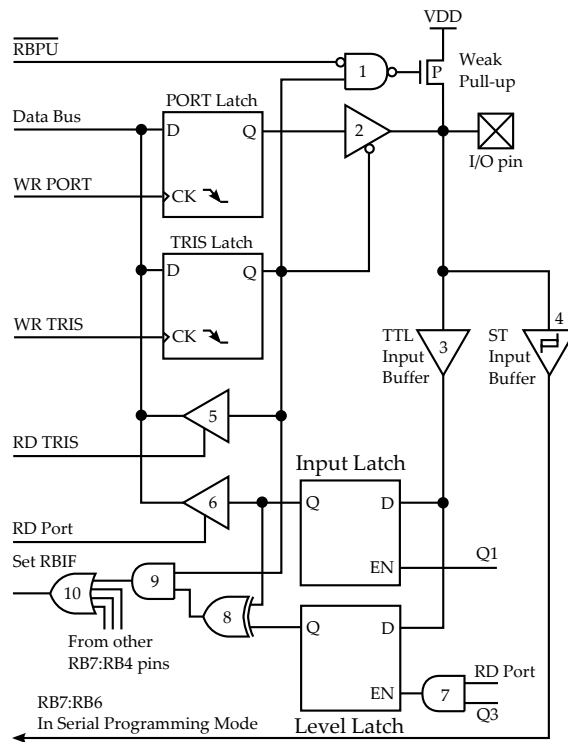



Figura 4.14: Architettura dei pin RB7:RB4

In essa si distingue il circuito di *pull up* nella parte superiore del circuito e, subito sotto, il circuito di uscita. Quest'ultimo è leggermente diverso da quello del PORTA dato che il pin di I/O è gestito direttamente da una porta *three state* e non da due MOS complementari distinti.

Sotto il TRISB Latch, responsabile della configurazione di I/O del singolo bit, si nota il circuito di input formato dalla porta TTL e dall'*Input Latch* e, sotto ancora, la gestione dell'*interrupt* relativa ai cambiamenti di stato delle linee di ingresso. Quest'ultima parte è probabilmente la più complessa, ma anche quella che rende il PORTB particolarmente versatile e funzionale.

Vediamo queste singole parti una ad una.

4.1.2.1 Il circuito di *pull up*

Il circuito di *pull up* è identico per tutti gli 8 bit del PORTB ed è evidenziato in fig. 4.14 a pagina 94.

Esso verrà trattato evidenziando la struttura dei 4 bit più significativi di detto port, ma le considerazioni che si faranno saranno identiche anche per i 4 bit meno significativi.

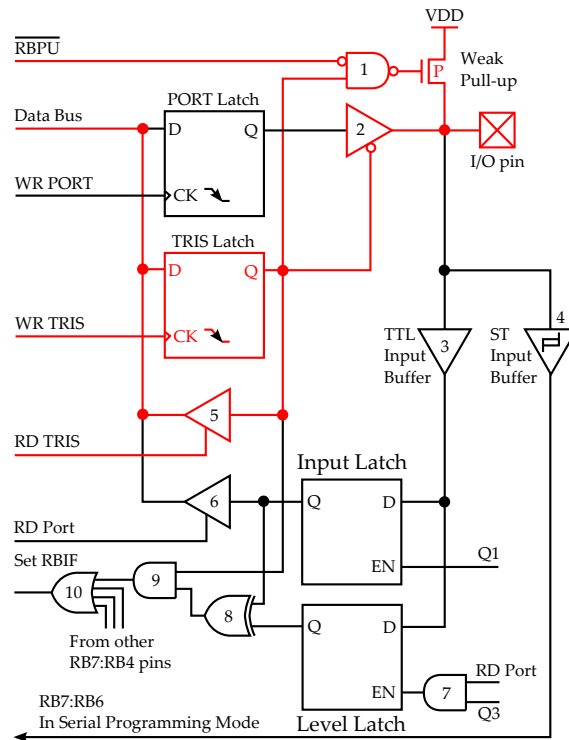



Figura 4.15: Circuito di *pull up*

Innanzi tutto è fondamentale, per poter parlare di *pull up*, che la linea sia programmata come ingresso. E' quindi necessario scrivere un 1 nel TRISB Latch. Tale valore logico disabilita l'uscita del buffer *three state* 2 e pone il pin di I/O in alta impedenza. In tal modo detto pin risulta essere flottante ed è possibile abilitare la modalità di *pull up* sull'intero port[6].

In queste condizioni, se bit di $\overline{RBPU}$ del registro delle opzioni (vedi sez.3.2) è attivo (ovvero a stato logico 0), la porta NAND 1 si ritrova con entrambi gli ingressi attivi, per cui può attivare la propria uscita ponendola a stato logico 0.

Tale stato logico, riportato sul *gate* del MOS a canale P lo attiva leggermente facendo scorrere una debole corrente da V_{DD} verso il pin di I/O, attivando in tal modo il *pull up*.

Lo stato del bit del TRISB Latch è sempre leggibile attraverso il buffer 5.

[6]Non è possibile abilitare il *pull up* solo per una singola linea: lo si può abilitare solamente per tutte le linee di ingresso del port oppure per nessuna.

4.1.2.2 Il circuito di uscita digitale dei pin RB7:RB4

In fig. 4.16 è evidenziato sommariamente il circuito di uscita digitale relativo alle linee RB7:RB4.

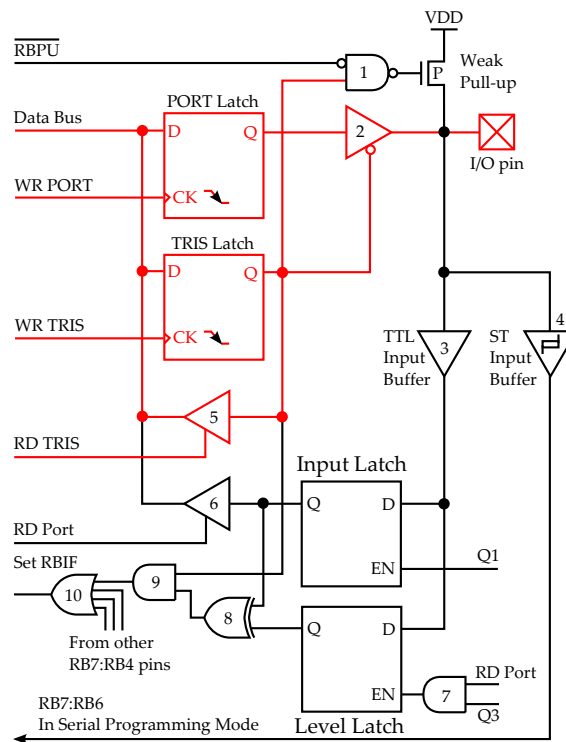



Figura 4.16: Architettura dell'uscita digitale dei pin RB7:RB4

Mediante il TRISB Latch è possibile, scrivendo uno 0, programmare la linea come uscita. Ciò abilita la porta *three state* 2 a porre in uscita sul pin di I/O lo stato dell'uscita Q del PORTB Latch.

Scrivendo, quindi, un 1 sul PORTB Latch si propaga sul pin di I/O uno stato logico 1 e scrivendo uno 0 sul PORTB Latch si propaga uno 0.

Contemporaneamente, l'uscita Q del TRISB Latch viene propagata sul secondo ingresso della porta NAND 2, che pone in uscita uno stato logico alto, interdicendo il MOS a canale P e disabilitando, in tal modo, il *pull up* sul pin di I/O.

Infine, è sempre possibile leggere lo stato logico del bit del TRISB Latch e dedurre se la linea è stata programmata in ingresso oppure in uscita.

Il PORTA Latch è leggibile mediante il circuito d'ingresso, che verrà descritto nella prossima sezione.

4.1.2.3 Il circuito di ingresso digitale dei pin RB7:RB4

In fig. 4.17 è evidenziato sommariamente il circuito di ingresso digitale relativo alle linee RB7:RB4.

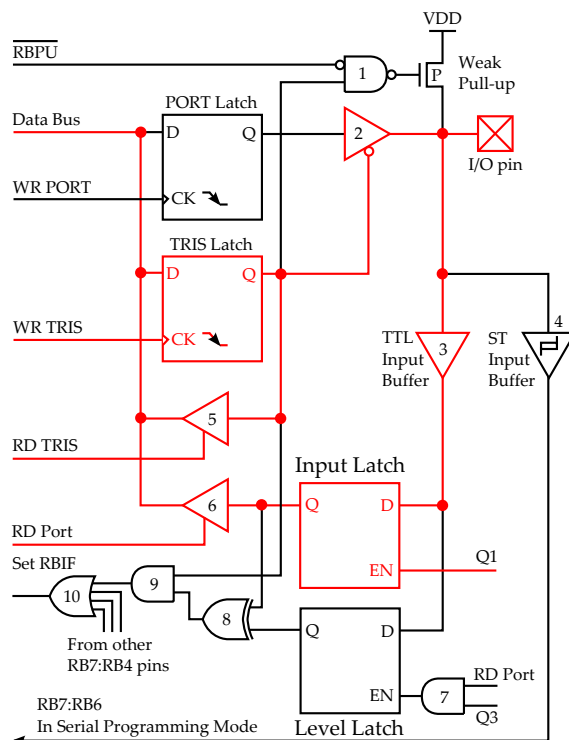



Figura 4.17: Architettura dell'ingresso digitale dei pin RB7:RB4

Affinchè il pin di I/O possa funzionare da ingresso è necessario che la porta *three state* 2 sia posta in alta impedenza. Ciò avviene mediante il TRISB Latch, la cui uscita Q viene posta a 1 quando si programma il pin di I/O come ingresso e disabilita in tal modo l'uscita del buffer 2.

Normalmente Q1[7] è in stato logico 0 e diventa 1 nel primo ciclo macchina di esecuzione dell'istruzione. In tale situazione l'*Input Latch* ha il segnale EN posto allo stato logico 1, per cui detto Latch è in *transparent mode* e il segnale presente all'ingresso D viene riproposto pari pari all'uscita Q, ovvero in ingresso buffer 6.

Quando il segnale RD Port diventa attivo, si disabilita lo stato di alta impedenza della porta 6 e il segnale presente sull'uscita Q dell'*Input Latch* viene posto sul *Data Bus*.

Si noti che la porta 3 è un semplice buffer TTL non triggerato, per cui il segnale d'ingresso presente al pin di I/O deve essere rigorosamente TTL compatibile, con fronti di salita e discesa ripidi.

Mediante la porta 5 è sempre possibile sapere se il pin di I/O è stato programmato come ingresso o come uscita.

[7]Per Q1 si intende quel segnale che viene attivato durante il primo ciclo macchina di esecuzione dell'istruzione. Vedi anche a tal proposito la sezione 4.1.6.

4.1.2.4 Il circuito di *interrupt-on-change*

In fig. 4.18 è evidenziato sommariamente il circuito di *interrupt-on-change*.

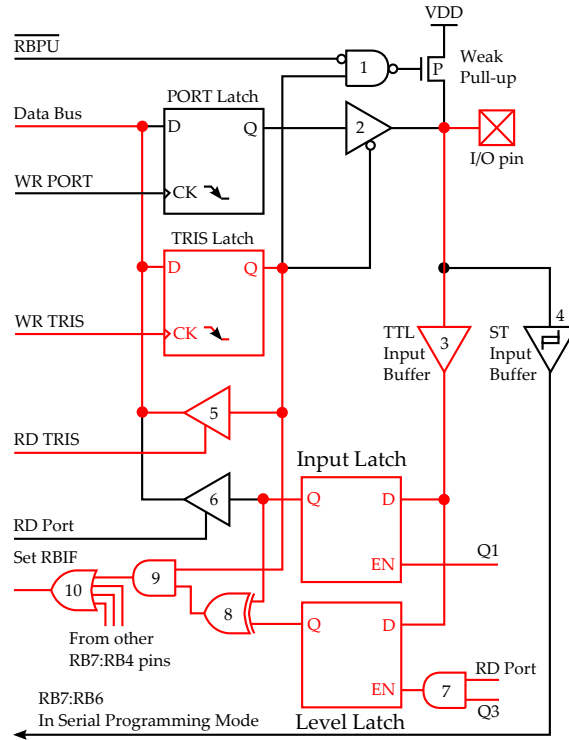



Figura 4.18: Architettura dell'*interrupt-on-change*

Affinché la modalità di *interrupt-on-change* possa essere utilizzata è necessario che sia attivo il circuito di ingresso. Esso viene evidenziato in figura solo per la parte strettamente necessaria, per non alimentare confusione, però, se ne deve tenere debito conto. Il TRIS Latch, quindi, avrà l'uscita Q posta a stato logico 1, che viene propagato anche sul primo ingresso della porta AND 9.

In tali condizioni si supponga di leggere lo stato logico del pin di I/O. Tale stato verrà memorizzato dall'*Input Latch* durante il ciclo di clock 1 (durante il quale si ha $Q1 = 1$) dell'esecuzione dell'istruzione di lettura e dal *Level Latch* durante il ciclo di clock 3 ($Q3 = 1$). Si vedano le figg. 4.18 e 4.19.

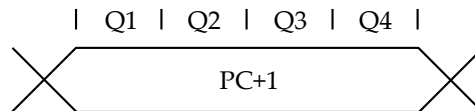



Figura 4.19: Impulsi di clock di un ciclo macchina

Da questo momento in poi, durante ogni ciclo di clock 1 delle successive istruzioni, verrà campionato lo stato del pin di I/O e verrà confrontato con lo stato logico memorizzato dal *Level Latch* attraverso la porta XOR 8.

Se la porta XOR rileverà una differenza fra i due livelli (quello campionato dall'*Input Latch* e quello memorizzato dal *Level Latch*) presenterà uno stato logico 1 in uscita, che verrà propagato fino all'uscita SET RBIF, attivando in tal modo il *flag* di interruzione e lanciando una procedura di *interrupt-on-change* se l'interruzione è abilitata.

4.1.2.5 Il circuito di *In Serial Programming*

In fig. 4.20 è evidenziato sommariamente il circuito di *In Serial Programming* utilizzante le linee RB7 e RB6.

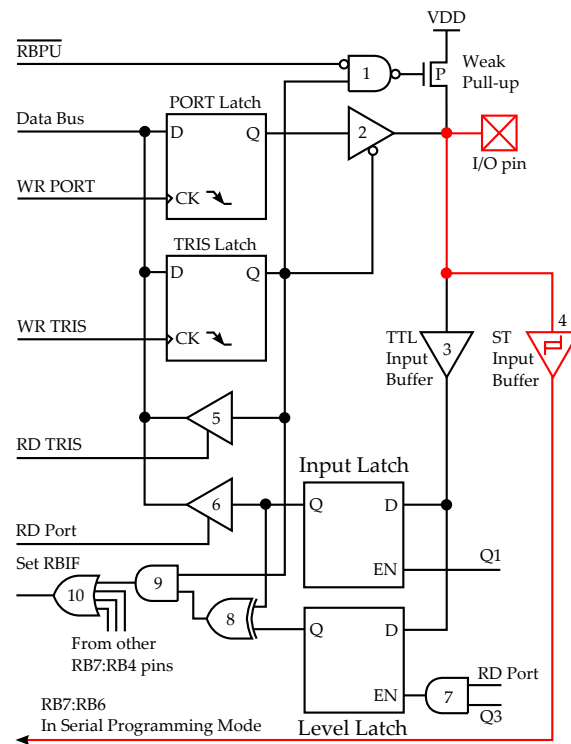



Figura 4.20: Architettura dell'*In Serial Programming*

Il circuito relativo alle linee RB7 e RB6 è formato semplicemente dalla porta triggerata 4, che ha il compito di "pulire" i segnali provenienti dall'emulatore prima di propagarli all'*In-Circuit Debugger* (vedi sez.1.10.4). Ciò si rende necessario perché, solitamente, detti segnali provenienti dal sistema di sviluppo sono veicolati da fili di 20-30cm di lunghezza e possono quindi fungere da antenna per eventuali radio disturbi.

La porta 4 provvede a rigenerare detti segnali.

4.1.2.6 L'architettura dei pin RB3:RB0

Nella fig. 4.21 è evidenziata l'architettura dei pin di I/O RB3:RB0 del PORTB.

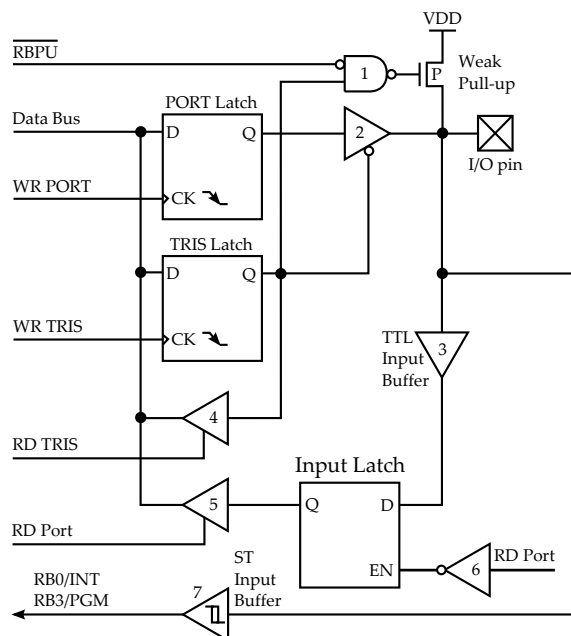



Figura 4.21: Architettura dei pin RB3:RB0

La struttura circuitale dei pin RB3:RB0 è decisamente più semplice di quella vista per i 4 bit più significativi del PORTB. Ciò evidenzia anche la minor potenza e flessibilità di tali bit del port.

Risulta mancante tutta la sezione relativa all'*interrupt-on-change* che, infatti, non appartiene ai bit RB3:RB0. Detti bit non sono, quindi, in grado di lanciare alcuna interruzione sul cambiamento di stato logico in ingresso. Tutto sommato non è grave che tale particolarità sia limitata ai soli 4 bit più significativi del PORTB, ma ciò implica una certa attenzione durante la progettazione *hardware* del sistema, in modo da non assegnare ai pin RB3:RB0 delle funzionalità che non hanno.

In compenso, la linea RB0/INT permette a segnali esterni, anche non rigorosamente digitali, di lanciare interruzione su un fronte a scelta. L'applicazione tipica di tale risorsa è la rilevazione di un segnale di *power fail* esterno: in caso di caduta di tensione da parte dell'alimentatore, si lancia un'interruzione attraverso il pin di RB0/INT per salvare, ad esempio, dei dati salienti in memoria non volatile, ad esempio la memoria Eeprom a bordo del microcontrollore, al fine di recuperarli alla prossima accensione.

Vediamo in dettaglio le varie sezioni del circuito di fig.4.21.

4.1.2.7 Il circuito di uscita digitale dei pin RB3:RB0

In fig. 4.22 è evidenziato sommariamente il circuito di uscita digitale relativo alle linee RB3:RB0.

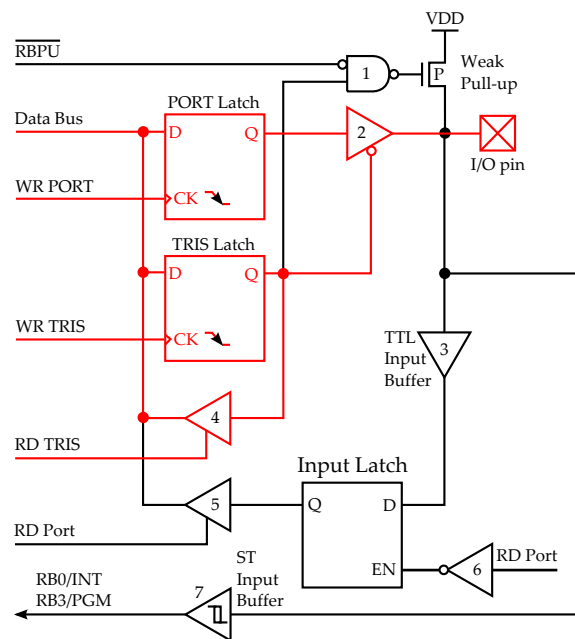



Figura 4.22: Architettura dell'uscita digitale dei pin RB3:RB0

Il circuito di fig. 4.22 dovrebbe essere ormai piuttosto familiare, per cui non sarà necessario trattare l'argomento da capo.

Per poter usufruire del pin di I/O come di un'uscita è necessario scrivere uno 0 logico nel Latch del TRISB. Tale stato logico abilita l'uscita del buffer 2, direttamente collegato al pin di I/O. In qualsiasi momento è possibile rileggere il settaggio del TRISB mediante la porta 4.

Il valore logico dell'uscita (ovvero del pin di I/O) è stabilito dal *PORTB Latch*, la cui uscita Q è collegata all'ingresso del buffer 2. Scrivendo uno 0 oppure un 1 nel Latch del PORTB tale valore verrà propagato sul pin di I/O attraverso, appunto, la porta 2.

Anche i bit RB3:RB0 sono dotati di *pull up*, esattamente come i bit RB7:RB3. Tale funzionalità è già stata trattata nella sezione 4.1.2.1, per cui si fa riferimento ad essa per eventuali dettagli. Si sottolinea ancora, invece, che il circuito di *pull up* non può essere sezionato essendo il segnale $\overline{RBPU}$ unico per tutto il PORTB. Quindi se tale bit viene attivato nel registro delle opzioni (vedi sez.3.2) la funzione di *pull up* viene attivata per tutti i bit del PORTB programmati come ingresso.

4.1.2.8 Il circuito di ingresso digitale dei pin RB3:RB0

In fig. 4.23 è evidenziato sommariamente il circuito di ingresso digitale relativo alle linee RB3:RB0.

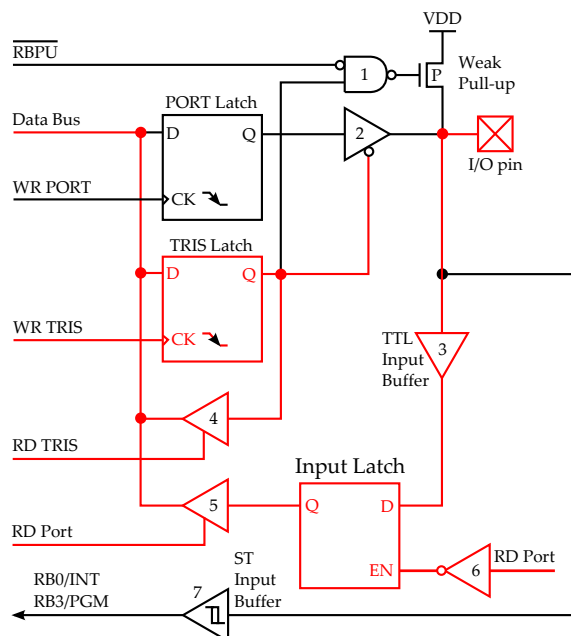



Figura 4.23: Architettura dell'ingresso digitale dei pin RB3:RB0

Anche il circuito d'ingresso delle linee RB3:RB0 è del tutto simile al circuito d'ingresso già visto, ad esempio, per le linee del PORTA. La parte relativa al *pull up* non viene evidenziata in fig. 4.23 per non creare confusione e perché detto circuito è già stato trattato nella sezione 4.1.2.1. A tutti gli effetti, però, può dirsi anch'esso facente parte del circuito d'ingresso.

Affinché il circuito d'uscita non interferisca con quello d'ingresso è necessario che l'uscita della porta 2 venga posta in alta impedenza. Ciò avviene automaticamente quando nel *TRISB Latch* si scrive un 1 logico al fine di programmare il pin di I/O come ingresso.

La lettura dello stato logico presente in ingresso avviene mediante lettura dell'*Input Latch*, attraverso il buffer TTL 3. Quando il segnale RD Port diventa attivo (alto) il Latch memorizza lo stato dell'ingresso presente sul pin di I/O e lo presenta all'ingresso del buffer 5. Quest'ultimo lo trasferisce sul bus dei dati.

La memorizzazione dello stato logico d'ingresso impedisce letture ambigue dello stesso.

In qualsiasi momento è sempre possibile conoscere lo stato memorizzato nel *TRISB Latch* attivando l'uscita del buffer 4 che presenta detto stato logico sul *Data Bus*.

4.1.2.9 Il circuito di ingresso dell'interruzione esterna

In fig. 4.24 è evidenziato il circuito di ingresso dell'interruzione esterna.

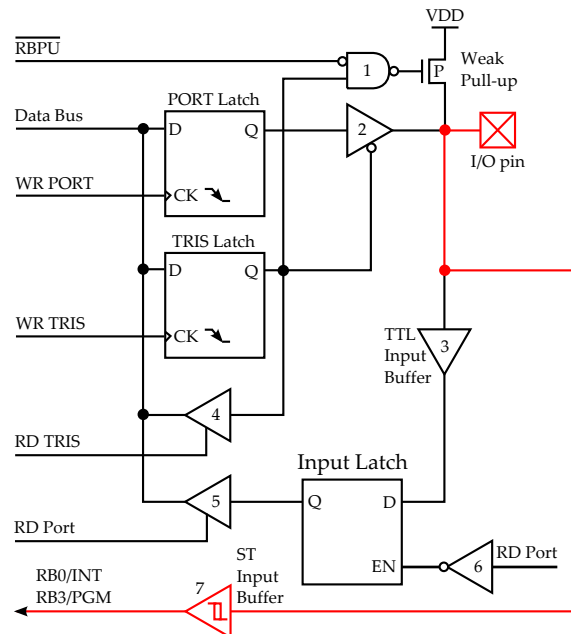



Figura 4.24: Architettura dell'ingresso dell'interruzione esterna

Il circuito è identico sia per il pin RB0/INT che per il pin RB3/PGM. Il fronte attivo di RB0/INT si imposta con il bit INTEDG del registro delle opzioni.

4.1.2.10 La programmazione del PORTB

La programmazione del PORTB presenta qualche differenza rispetto a quella del PORTA. In fig. 4.25 sono riassunti i registri coinvolti.

Address	Name	bit 7	bit 6	bit 5	bit 4	bit 3	bit 2	bit 1	bit 0	Value on POR, BOR	Value on all other RESETS
06h, 106h	PORTB	RB7	RB6	RB5	RB4	RB3	RB2	RB1	RB0	xxxx xxxx	uuu uuuu
86h, 186h	TRISB	PORTB Data Direction Register								1111 1111	1111 1111
81h, 181h	OPTION_REG	RPBU	INTEDG	T0CS	T0SE	PSA	PS2	PS1	PS0	1111 1111	1111 1111
0Bh, 10Bh 8Bh, 18Bh	INTCON	GIE	PEIE	T0IE	INTE	RBIE	T0IF	INTIF	RBIF	0000 000x	0000 000u

Legend: x = unknown, u = unchanged, / = unimplemented locations read as 0
Shaded cells are not used by PORTB

Figura 4.25: Registri coinvolti nella programmazione del PORTB

Si noti come all'accensione il valore logico dei bit RB7:RB0 sia impredicabile. Inoltre, tutti gli ingressi sono protetti mediante diodi come nel PORTA.

I registri coinvolti nella programmazione del PORTB come port di I/O, quindi senza il coinvolgimento di altre periferiche o altre funzionalità diverse dal semplice I/O di dati digitali, sono, oltre agli ovvi PORTB e TRISB, anche l'INTCON e l'OPTION_REG.

La Microchip fornisce esempi di inizializzazione del PORTB, che però presuppongono un reset. A volte è utile reinizializzare il PORTB per poterlo usare come port di I/O senza poter lanciare un reset.

Si fornisce, quindi, di seguito un'inizializzazione del PORTB di carattere generale, utilizzabile sempre, che non utilizza alcuna funzione aggiuntiva oltre all'I/O (nessun *pull up*, *interrupt-on-change*, ecc.).

Il codice è il seguente:

Listing 4.2: Inizializzazione PORTB

```
banksel PORTB           ;Select Bank 0
clrf    PORTB           ;Initialize PORTB by clearing
                        ;output data latches
banksel TRISB
bsf     OPTION_REG,RBPU ;No pull-ups
bcf     INTCON,INTE     ;Disables external interrupt
bcf     INTCON,INTIF    ;Cleares external interrupt
bcf     INTCON,RBIE     ;Disables interrupt-on-change
bcf     INTCON,RBIF     ;Cleares interrupt-on-change
movlw   0xCF            ;Initialize TRISB
movwf   TRISB           ;Set RB<3:0> as inputs
                        ;and RB<7:4> as outputs.
```

Si noti come prima si azzeri il PORTB e poi si programmi il TRISB. Tale pratica, usata anche nell'inizializzazione del PORTA, è bene venga sempre attuata.

Inoltre, sono interessanti anche le istruzioni che coinvolgono il registro di controllo delle interruzioni e delle due coppie di bit `INTE`, `INTIF` e `RBIE` e `RBIF`. Infatti viene prima azzerato il bit di abilitazione dell'interruzione e dopo viene azzerato l'eventuale *flag* di notifica.

Questo perché, potrebbe succedere che, azzerando prima il bit di notifica e disabilitando poi l'interruzione, potrebbe verificarsi un'interruzione proprio fra dette due istruzioni, con effetti spiacevoli: il programmatore potrebbe convincersi di aver risolto i suoi problemi con l'interruzione esterna proprio nel momento in cui ne viene lanciata una!

Si noti anche, infine, che nella seconda parte del codice viene effettuata una seconda selezione di banco con TRISB come operando. In realtà poi si accede al registro OPTION_REG e INTCON, che però sono entrambi raggiungibili dal banco 1. Un siffatto codice è molto compatto ma richiede una certa conoscenza della mappatura dei registri o, perlomeno, una certa attenzione al problema.

4.1.3 Il PORTC

Il PORTC è un port bidirezionale a 8 bit. Si tratta di un port coinvolto in maniera piuttosto importante con le periferiche esterne. Infatti, il PORTC:

- permette la gestione dell'oscillatore del Timer 1;
- pone in uscita, se selezionati e abilitati, i segnali del PWM1 e PWM2;
- gestisce i segnali della linea seriale sincrona;
- gestisce i segnali della linea seriale asincrona.

Tutte le linee del port sono multiplexate con linee di periferiche. La tabella riassuntiva delle funzioni abbinate ai singoli pin è la seguente:

Name	Bit#	Buffer	Function
RC0/T1OSO/T1CKI	bit0	ST	Input/output port pin or Timer1 oscillator output/Timer1 clock input.
RC1/T1OSI/CCP2	bit1	ST	Input/output port pin or Timer1 oscillator input or Capture2 input/-Compare2 output/PWM2 output.
RC2/CCP1	bit2	ST	Input/output port pin or Capture1 input/Compare1 output/P-WM1 output.
RC3/SCK/SCL	bit3	ST	RC3 can also be the synchronous serial clock for both SPI and I^2C modes.
RC4/SDI/SDA	bit4	ST	RC4 can also be the SPI Data In (SPI mode) or data I/O (I^2C mode).
RC5/SDO	bit5	ST	Input/output port pin or Synchronous Serial Port data output.
RC6/TX/CK	bit6	ST	Input/output port pin or USART Asynchronous Transmit or Synchronous Clock.
RC7/RX/DT	bit7	ST	Input/output port pin or USART Asynchronous Receive or Synchronous Data.

Tabella 4.3: Tipologia delle linee del PORTC

Una prima particolarità del PORTC è che tutti gli ingressi sono *Schmitt Trigger*, il che lo rende prezioso in presenza di segnali digitali non troppo puliti.

I bit RC0 e RC1 possono essere usati per collegare un quarzo che scandisca il clock del Timer 1.

I successivi due bit possono essere usati come le uscite del PWM1 e del PWM2.

I segnali RC3, RC4 e RC5 possono essere usati per gestire una linea seriale sincrona quale la I^2C o la SPI[8].

I segnali RC6 e RC7 possono gestire, invece, una linea seriale asincrona.

[8]Gli standard di comunicazione seriale sincrona SPI e I^2C verranno trattati nella sezione **??**.

4.1.3.1 L'architettura dei pin RC7:RC5 e RC2:RC0

Nella fig. 4.26 è evidenziata l'architettura dei pin di I/O RC7:RC5 e RC2:RC0 del PORTC.

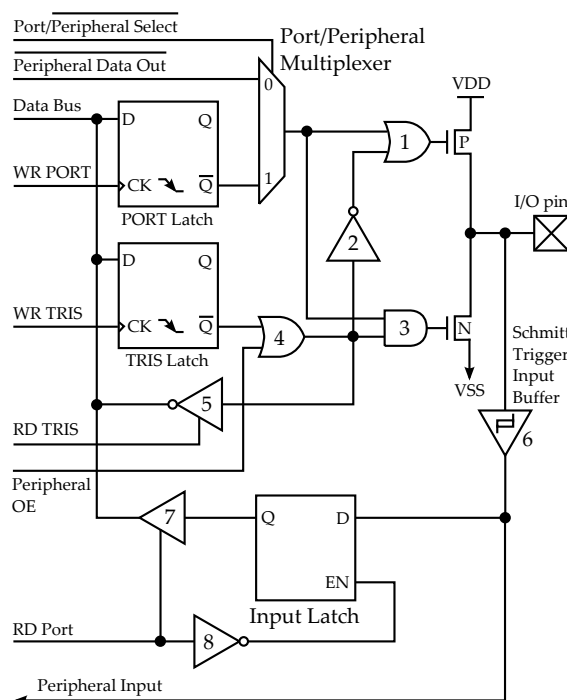



Figura 4.26: Architettura dei pin RC7:RC5 e RC2:RC0

L'architettura appare leggermente più complessa delle precedenti. Ciò è dovuto al numero e alla varietà delle periferiche coinvolte. La circuiteria relativa ai Latch è fondamentalmente la solita. Cambia in maniera piuttosto evidente la circuiteria che pilota il pin di I/O, essendo più complessa.

Inoltre, nella parte alta del circuito appare il simbolo di un multiplexer, indicato come *Port/Peripheral Multiplexer*. Esso ha il compito di selezionare il segnale da porre in uscita al pin di I/O: quello proveniente dal port di I/O oppure quello proveniente dalla periferica.

La complessità, quindi, appare circoscritta al solo circuito di uscita, lasciando il circuito di ingresso sostanzialmente simile a quello dei due port precedentemente visti.

Infine, pur non essendo evidenziato esplicitamente nel circuito di fig.4.26, tutti gli ingressi del PORTC (compresi quindi gli ingressi RC4:RC3) sono protetti dai diodi di *clamping* posti fra V_{SS} e V_{DD} (vedi la sezione 4.1.1.10). In tal modo gli ingressi sono protetti da tensioni maggiori di V_{DD} oppure minori di V_{SS}.

4.1.3.2 Il circuito di selezione *Port/Peripheral*

In fig. 4.27 è evidenziato sommariamente il circuito di selezione fra port di I/O e periferica.

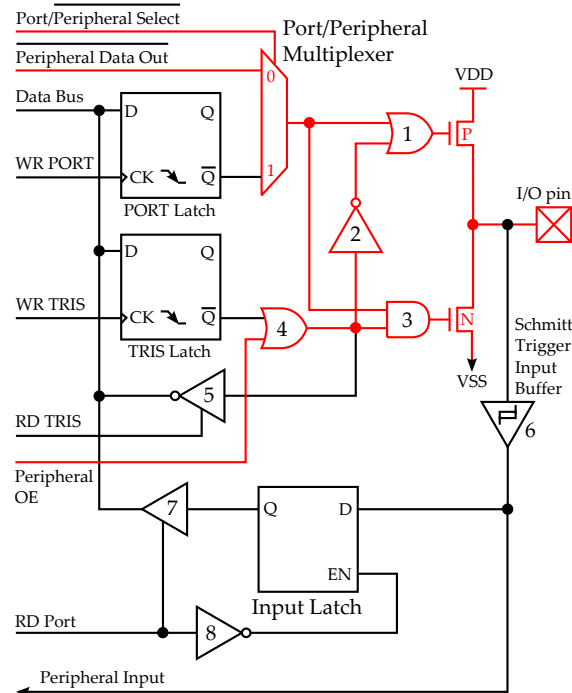



Figura 4.27: Il circuito di selezione *Port/Peripheral*

Il $Port/\overline{PeripheralMultiplexer}$ ha il compito, quando il pin è programmato in uscita, di selezionare quale dato porre sul pin di I/O: quello del PORTC oppure quello della periferica associata al pin in questione (vedi, a tal proposito la tabella 4.3 a pagina 105).

Quindi si può supporre che nel *TRIS Latch* sia stato memorizzato uno 0 logico. Quindi l'uscita negata, collegata al primo ingresso della porta OR 4, avrà valore logico 1. Ciò comporta un 1 logico sul secondo ingresso della porta AND 3 e uno 0 logico sul secondo ingresso della porta OR 1: il valore di uscita di dette porte dipenderà, quindi, dal valore del rispettivo primo ingresso, ovvero dall'uscita del multiplexer. Si noti, però, che il livello logico che il pin di I/O assume è la *negazione* dell'uscita del multiplexer.

Quando la periferica associata al pin in questione è disabilitata, il segnale di selezione $Port/\overline{PeripheralSelect}$ viene posto a livello logico 1 e il multiplexer pone sul primo ingresso della porta OR 1 e AND 3 il valore logico presente su $\overline{Q}$ del *Port Latch*. Quando, invece, il segnale di selezione del multiplexer è posto a 0, esso pone sulla sua uscita il segnale $\overline{PeripheralDataOut}$[9].

[9] Si noti che in alcuni *datasheet* ufficiali della Microchip il segnale $\overline{PeripheralDataOut}$ appare non negato. Ciò è *errato* dato che l'uscita del multiplexer viene invertita prima di essere posta sul pin di I/O.

Il segnale *Peripheral OE* viene attivato solo se $Port/\overline{PeripheralSelect} = 0$. Di ciò si deve tenere conto, dato che alcune periferiche possono sovrascrivere il valore del *TRIS Latch*. L'argomento verrà ripreso nella sezione 4.1.3.9.

4.1.3.3 Il circuito di uscita digitale del PORTC

In fig. 4.28 è evidenziato sommariamente il circuito di uscita digitale relativo alle linee RC7:RC5 e RC2:RC0.

Figura 4.28: Architettura dell'uscita digitale dei pin RC7:RC5 e RC2:RC0

Il circuito di selezione del segnale di uscita (negato) è già stato illustrato nella sezione precedente, per cui tale parte non è stata evidenziata in fig. 4.28.

Tolta detta parte, il circuito è molto simile ai precedenti, con il *TRISC Latch* che memorizza il tipo di linea (ingresso o uscita) e con il *PORTC Latch* che memorizza lo stato logico che il pin di I/O deve assumere.

L'unica differenza è data dalla presenza della porta OR 4, che ha il compito di propagare lo stato logico 1 all'ingresso della porta AND 3.

Si noti la somiglianza del circuito di uscita con quello del PORTA, mentre i segnali di uscita dei latch risultano essere invertiti rispetto a quelli del PORTB.

Ciò è dovuto al circuito di pilotaggio del pin di I/O: nel caso del PORTA esso inverte i segnali provenienti dai due latch, mentre nel caso del PORTB detti segnali non vengono invertiti.

Mediante la porta *three state* 5 è possibile, come nei precedenti casi, leggere lo stato memorizzato nel *TRISC Latch*.

4.1.3.4 Il circuito di ingresso digitale del PORTC

In fig. 4.29 è evidenziato sommariamente il circuito di ingresso digitale relativo alle linee RC7:RC5 e RC2:RC0.

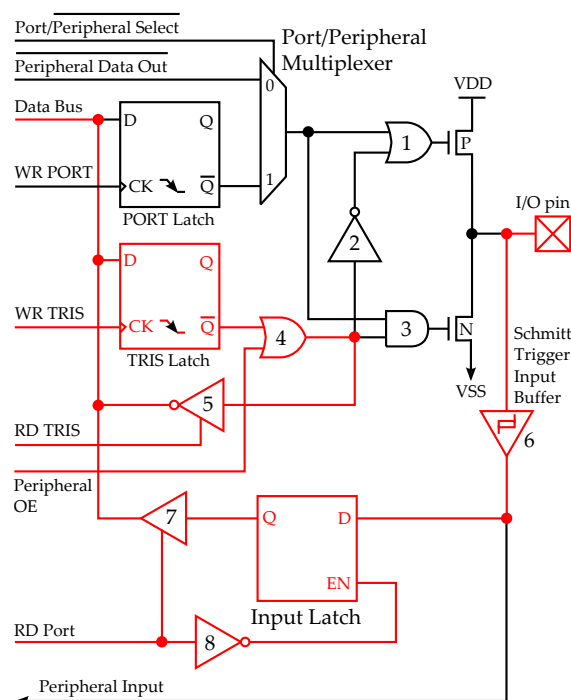



Figura 4.29: Architettura dell'ingresso digitale dei pin RC7:RC5 e RC2:RC0

Anche in questo caso il circuito di ingresso è molto simile ai precedenti. Si rilevano due aspetti di una certa importanza: gli ingressi sono tutti di tipo *Schmitt Trigger* e il valore di uscita del *TRISC Latch* può essere sovrascritto dal segnale *Peripheral OE*.

Il primo aspetto permette la lettura in ingresso di segnali i cui fronti non siano ripidissimi e quindi potenzialmente fonte di problemi se letti come segnali digitali. Il progettista ne deve tenere conto utilizzando correttamente detta risorsa.

Il secondo aspetto può essere fonte di problemi in fase di programmazione. La porta NOT 5 ha il compito di porre sul bus dei dati lo stato logico abbinato alla programmazione del pin di I/O (ovvero al suo stato di ingresso o uscita). Detta porta non ha, però, al suo ingresso il valore logico dell'uscita $\overline{Q}$ del *TRISC Latch*, bensì quello della porta OR 4. Ciò comporta che se si programma il *TRISC Latch* con il valore 1 (pin di I/O programmato come ingresso) e poi si seleziona una periferica che utilizza detto pin come uscita[10], il valore letto dalla porta NOT 5 va interpretato: se si legge tale valore quando la periferica non abilita il proprio segnale di *Peripheral OE* si legge un 1 logico, altrimenti si legge uno 0 logico. L'argomento verrà approfondito nella sezione 4.1.3.9.

[10] E' il caso, ad esempio, del pin RC5, che è abbinato al segnale SDO (uscita) della porta seriale sincrona.

4.1.3.5 Il circuito di *Peripheral Input*

In fig. 4.30 è evidenziato sommariamente il circuito di ingresso della periferica relativo alle linee RC7:RC5 e RC2:RC0.

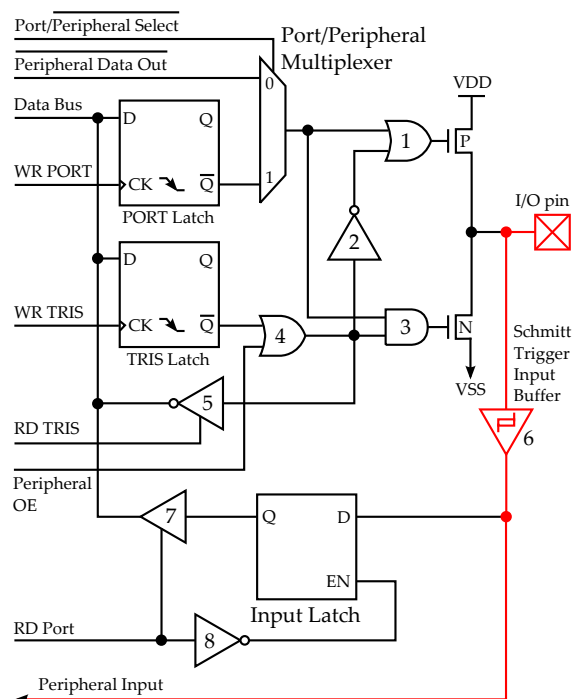



Figura 4.30: Architettura del *Peripheral Input*

Il circuito evidenziato in fig. 4.30 va inteso in maniera piuttosto elastica. Nella sezione **??** verrà presentata l'architettura del Timer 1, che utilizza i pin RC0 e RC1, dove le porte triggerate *non compaiono* (oppure compaiono nel posto sbagliato).

C'è da supporre, quindi, che il circuito d'ingresso verso le periferiche relativo a detti pin sia leggermente diverso da quello evidenziato in fig. 4.30. Con ogni probabilità, nel caso in cui RC0 e RC1 siano utilizzati come ingressi per l'oscillatore del Timer 1, la porta triggerata non è utilizzata, nè collegata come in figura.

4.1.3.6 L'architettura dei pin RC4:RC3

La struttura dell'architettura relativa ai bit RC4:RC3 del port è leggermente più complicata rispetto ai restanti pin. Ciò è dovuto ad una tecnologia di ingresso più versatile, in modo da poter essere in armonia sia con lo standard I^2C che con quello SMBus.

Il SMBus è del tutto simile all'I^2C bus, anzi, in maniera piuttosto impropria si potrebbe sostenere che ne è un sottoinsieme.

Il protocollo del bus verrà studiato nella sezione **??**, per ora sia sufficiente sapere che la principale differenza fra i due bus è data dai livelli di tensione. Senza entrare nei dettagli, supponendo una tensione di alimentazione di 5V, si possono distinguere nei due bus i seguenti valori:

Bus	High Value	Low Value
I^2C	3.0V	1.5V
SMBus	2.1V	0.8V

Tabella 4.4: Bus I^2C e SMBus a confronto

La tabella 4.4 spiega parte dell'architettura dei pin RC4:RC3 che verrà illustrata tra breve.

Nella fig. 4.31 è evidenziata l'architettura dei pin RC4:RC3 del PORTC.

Figura 4.31: Architettura dei pin RC4:RC3

Nella parte destra del circuito si notano le due porte triggerate che hanno il compito di leggere il segnale presente al pin di I/O con due differenti coppie di livelli di tensione: uno osservante dello standard I^2C ed uno osservante dello standard SMBus.

La restante parte del circuito è identica a quella già descritta per i pin RC7:RC5 e RC2:RC0, per cui ci si limiterà a trattare il solo circuito di ingresso del segnale di periferica.

4.1.3.7 Il circuito di I^2C/SMBus Peripheral Input

In fig. 4.32 è evidenziato sommariamente il circuito di ingresso della periferica I^{2}C/SMBus relativo alle linee RC4:RC3.

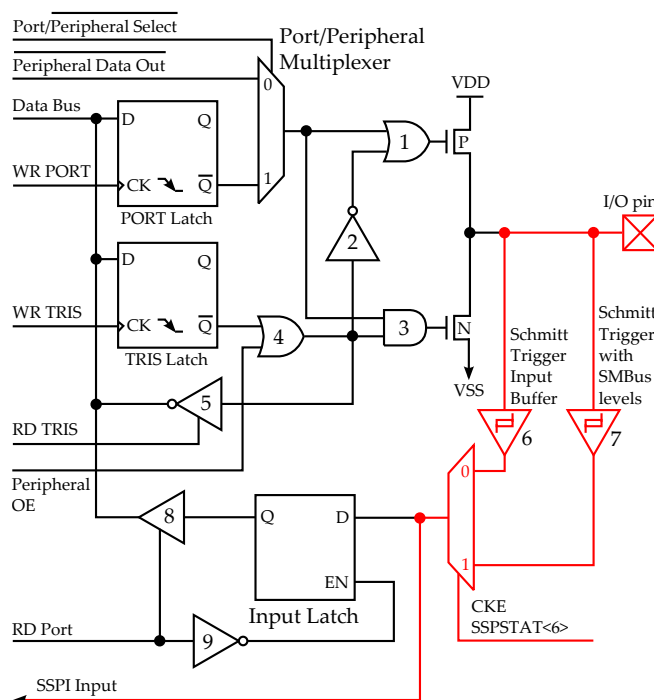



Figura 4.32: Architettura dell'*I^{2}C/SMBus Peripheral Input*

Il buffer triggerato 6 ha livelli d'ingresso compatibili con lo standard I^{2}C, avendo esso i due livelli di soglia posti a $1/3V_{DD}$ e $2/3V_{DD}$. Il buffer triggerato 7, invece ha due livelli più bassi dovendo esso discriminare fra una fascia di tensioni compatibile con lo 0 logico (0-0.8V) ed una con lo stato logico 1 (2.1-5V).

La selezione della porta triggerata viene fatta attraverso il bit `CKE` del registro SSPSTAT: se posto a 1 viene selezionata la porta compatibile allo standard SMBus altrimenti quella compatibile con l'I^{2}C bus.

4.1.3.8 La programmazione del PORTC

Anche la programmazione del PORTC presenta delle caratteristiche sue proprie. Essendo coinvolte ben cinque tipi di periferiche diverse, l'inizializzazione deve essere effttuata con maggior attenzione. I motivi di certi settaggi verranno compresi quando si affronterà lo studio della periferica coinvolta.

In fig. 4.33 a fronte sono evidenziati i registri coinvolti nella programmazione del PORTC. I bit colorati di grigio non sono direttamente coinvolti in detta inizializzazione.

Essendo cinque le periferiche interessate, sono anche cinque i registri coinvolti, oltre al PORTC e al TRISC. Esaminiamoli uno ad uno.

Address	Name	bit 7	bit 6	bit 5	bit 4	bit 3	bit 2	bit 1	bit 0	Value on POR, BOR	Value on all other RESETS
07h, 107h	PORTC	RC7	RC6	RC5	RC4	RC3	RC2	RC1	RC0	xxxx xxxx	uuu uuuu
87h, 187h	TRISC	PORTC Data Direction Register								1111 1111	1111 1111
10h	T1CON	/	/	T1CKPS1	T1CKPS0	T1OSCEN	T1SYNC	TMR1CS	TMR1ON	--00 0000	--00 0000
17h	CCP1CON	/	/	CCP1X	CCP1Y	CCP1M3	CCP1M2	CCP1M1	CCP1M0	--00 0000	--00 0000
1Dh	CCP2CON	/	/	CCP2X	CCP2Y	CCP2M3	CCP2M2	CCP2M1	CCP2M0	--00 0000	--00 0000
14h	SSPCON	WCOL	SSPOV	SSPEN	CKP	SSPM3	SSPM2	SSPM1	SSPM0	0000 0000	0000 0000
18h	RCSTA	SPEN	RX9	SREN	CREN	ADDEN	FERR	OERR	RX9D	0000 000x	0000 000u

Legend: x = unknown, u = unchanged, / = unimplemented locations read as 0
Shaded cells are not used by PORTB

Figura 4.33: Registri coinvolti nella programmazione del PORTC

Il registro T1CON è il registro di controllo del Timer 1. Il bit interessato è il `T1OSCEN`, che permette l'abilitazione o meno dell'oscillatore esterno del Timer 1. Tale bit va posto a 0 logico se si vuole usare il PORTC solo come I/O.

I registri CCP1CON e CCP2CON sono i registri d controllo dei due moduli CCP 1 e 2. Per disabilitarli si devono azzerare i rispettivi 4 bit meno significativi.

Il registro SSPCON controlla il modulo di comunicazione seriale sincrona (mediante SPI oppure I^2C). La disabilitazione di tale modulo si ottiene azzerando il bit `SSPEN`.

Infine, il registro RCSTA contiene il bit di abilitazione del modulo di comunicazione seriale asincrona: `SPEN`. Anche tale bit va azzerato.

Il codice di inizializzazione del PORTC, nel caso in cui esso debba essere usato solo come port di I/O è il seguente:

Listing 4.3: Inizializzazione PORTC

```
banksel PORTC           ;Select Bank 0
clrf    PORTC           ;Initialize PORTC by clearing
                        ;output data latches
bcf     T1CON,T1OSCEN   ;No external oscillator
clrf    CCP1CON         ;Disables CCP1 module
clrf    CCP2CON         ;Disables CCP2 module
bcf     SSPCON,SSPEN    ;Disables SPI module
bcf     RCSTA,SPEN      ;Disables USART module
banksel TRISC
movlw   0xCF            ;Initialize TRISC
movwf   TRISC           ;Set RC<3:0> as inputs
                        ;and RC<7:4> as outputs.
```

4.1.3.9 Le periferiche e il TRISC Latch

A causa delle particolari periferiche gestite attraverso il PORTC, si deve prestare particolare attenzione alla programmazione di I/O.

Alcune periferiche "vedono" il corrispondente pin o solo come ingresso o solo come uscita, per cui sarebbe bene che anche il *TRISC Latch* si uniformasse a tale limite.

Ad esempio, se si decide di usare il modulo CCP1 e si sceglie la modalità *Capture Mode*, il bit RC2 verrà programmato automaticamente in ingresso dalla periferica stessa. Se invece si sceglie il modulo comparatore o PWM, il bit RC2 viene automaticamente programmato come uscita. E' bene che il programmatore si adegui programmando in analogo modo anche il latch del TRISC, in modo da non effettuare letture del tipo di I/O prive di senso.

La Microchip sconsiglia anche di programmare il TRISC con le istruzioni `bsf`, `bcf` e `xorwf`, che sono delle istruzioni *read-modify-write*: nel caso in cui la relariva periferica sia abilitata, dette istruzioni possono produrre errori.

4.1.4 Il PORTD

Il PORTD è un port bidirezionale a 8 bit. Esso è presente solamente nelle versioni a 40 pin della famiglia PIC16F87x, ossia nel PIC16F874 e nel PIC16F877. Si tratta di un port meno coinvolto con le periferiche esterne rispetto al PORTC. Infatti, il PORTD prevede solamente, oltre alla gestione dell'I/O, il solo uso del *Parallel Slave Port* (PSP).

La tabella riassuntiva delle funzioni abbinate ai singoli pin appare, quindi, piuttosto semplice ed è la seguente:

Name	Bit#	Buffer	Function
RD0/PSP0	bit0	ST/TTL	Input/output port pin or parallel slave port bit0.
RD1/PSP1	bit1	ST/TTL	Input/output port pin or parallel slave port bit1.
RD2/PSP2	bit2	ST/TTL	Input/output port pin or parallel slave port bit2.
RD3/PSP3	bit3	ST/TTL	Input/output port pin or parallel slave port bit3.
RD4/PSP4	bit4	ST/TTL	Input/output port pin or parallel slave port bit4.
RD5/PSP5	bit5	ST/TTL	Input/output port pin or parallel slave port bit5.
RD6/PSP6	bit6	ST/TTL	Input/output port pin or parallel slave port bit6.
RD7/PSP7	bit7	ST/TTL	Input/output port pin or parallel slave port bit7.

Tabella 4.5: Tipologia delle linee del PORTD

Si noti che il pin di I/O, quando è programmato come ingresso, può essere di tipo TTL oppure *Schmitt Trigger*. Anche se dalla fig. 4.34 a pagina 115 ciò non appare, quando il PORTD è usato come port di I/O gli ingressi sono di tipo *Schmitt Trigger*, mentre quando il PORTD è usato come *Parallel Slave Port* gli ingressi sono di tipo TTL.

Il *Parallel Slave Port* non è frequentemente usato a causa dell'eccessivo impegno di risorse. Un'applicazione tipica è data dall'interfaccia verso memorie esterne (retaggio storico, più che altro) oppure verso convertitori analogico-digitali.

In questo secondo caso l'interfaccia parallela ha maggior senso, soprattutto nel caso in cui si utilizzino convertitori di tipo *flash*, dato che un'interfaccia seriale potrebbe essere troppo lenta e rendere vana la velocità di conversione del convertitore *flash*.

Dal punto di vista circuitale, invece, il PORTD non aggiunge nulla di più di quanto già visto nei precedenti port. Si tratta senz'altro della più semplice delle versioni circuitali fin qui vista, per cui si ritiene sufficiente esporre solamente l'architettura del port, senza doverla studiare in dettaglio.

4.1.4.1 L'architettura del PORTD

Nella fig. 4.34 è evidenziata l'architettura dei pin di I/O del PORTD.

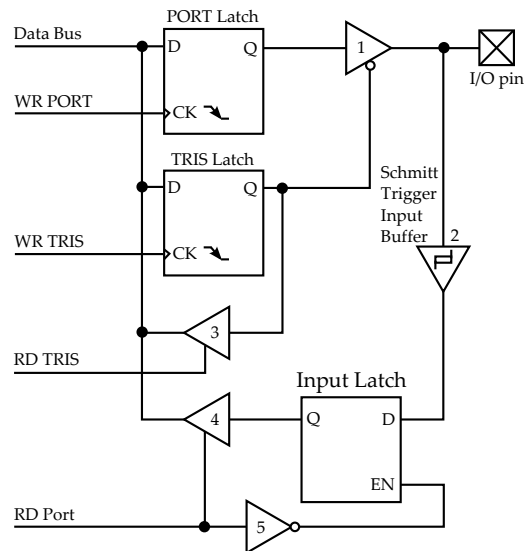



Figura 4.34: Architettura del PORTD

Come si vede il circuito è assolutamente simile alla parte comune dei port visti precedentemente. Non si ritiene quindi necessario studiare le singole parti che lo compongono. Unico dettaglio degno di nota: in figura la porta 2 è indicata come porta triggerata, mentre sappiamo che ciò è vero solo se il PORTD

viene selezionato come port di I/O. Se viene, invece, selezionato il PSP, le porte di ingresso sono con livelli di soglia TTL.

Anche i pin di I/O del PORTD sono protetti contro le tensioni maggiori di V_{DD} e minori di V_{SS}.

4.1.4.2 La programmazione del PORTD

L'inizializzazione del PORTD è piuttosto semplice, dato che solamente il *Parallel Slave Port* è coinvolto.

In fig. 4.35 a pagina 116 sono evidenziati i registri che influenzano la programmazione del PORTD.

Address	Name	bit 7	bit 6	bit 5	bit 4	bit 3	bit 2	bit 1	bit 0	Value on POR, BOR	Value on all other RESETS
08h, 108h	PORTD	RD7	RD6	RD5	RD4	RD3	RD2	RD1	RD0	xxxx xxxx	uuu uuuu
88h, 188h	TRISD	PORTD Data Direction Register								1111 1111	1111 1111
89h, 189h	TRISE	IBF	OBF	IBOV	PSPMODE	/	PORTE Data Direction bits			0000 -111	0000 -111

Legend: x = unknown, u = unchanged, / = unimplemented locations read as 0
Shaded cells are not used by PORTD

Figura 4.35: Registri coinvolti nella programmazione del PORTD

Il programma di prima inizializzazione del PORTD diventa quindi il seguente:

Listing 4.4: Inizializzazione PORTD

```
banksel PORTD    ;Select Bank 0
clrf    PORTD    ;Initialize PORTD by clearing
                 ;output data latches
banksel TRISD
bcf     TRISE    ;Disables PSP mode
movlw   0xCF     ;Initialize TRISD
movwf   TRISD    ;Set RD<3:0> as inputs
                 ;and RD<7:4> as outputs.
```

Si noti che i registri TRISD e TRISE sono posti nello stesso banco.

4.1.5 Il PORTE

Il PORTE è un port bidirezionale a soli 3 bit. Esso è presente solamente nelle versioni a 40 pin della famiglia PIC16F87x, ossia nel PIC16F874 e nel PIC16F877.

Oltre che ai pin di I/O, esso ospita anche i tre bit di *handshake* della periferica PSP e i restanti tre bit di ingresso analogico. Quindi i PIC16F87x equipaggiati con 28 pin possiedono un convertitore analogico-digitale a 5 canali (AN4:AN0),

mentre quelli equipaggiati con 40 pin possiedono un convertitore a 8 canali (AN7:AN0).

I segnali di *handshake* del *Parallel Slave Port* sono quelli classici: $\overline{RD}$, $\overline{WR}$ e $\overline{CS}$, tutti attivi bassi.

La tabella riassuntiva delle funzioni abbinate ai singoli pin appare, quindi, piuttosto semplice ed è la seguente:

Name	Bit#	Buffer	Function
RE0/$\overline{RD}$/AN5	bit0	ST/TTL	I/O port pin or read control input in Parallel Slave Port mode or analog input: $\overline{RD}$ 1 = Idle 0 = Read operation. Contents of PORTD register are output to PORTD I/O pins (if chip selected).
RE1/$\overline{WR}$/AN6	bit1	ST/TTL	I/O port pin or write control input in Parallel Slave Port mode or analog input: $\overline{WR}$ 1 = Idle 0 = Write operation. Value of PORTD I/O pins is latched into PORTD register (if chip selected).
RE2/$\overline{CS}$/AN7	bit2	ST/TTL	I/O port pin or chip select control input in Parallel Slave Port mode or analog input: $\overline{CS}$ 1 = Device is not selected 0 = Device is selected

Tabella 4.6: Tipologia delle linee del PORTE

I tre segnali di *handshake* presenti nel PORTE sono pensati per interfacciare il PORTD con il bus dati di un dispositivo parallelo esterno, che potrebbe essere una memoria, un microprocessore, un ADC, ecc. Per poter sfruttare le funzionalità offerte dal PSP è, però, necessario che l'interfaccia esterna sia compatibile con quella proposta dal PIC.

La circuiteria, i livelli attivi e la dinamica dei segnali è comunque di tipo classico, per cui la probabilità di un interfacciamento diretto e senza problemi è molto alta.

4.1.5.1 L'architettura del PORTE

L'architettura proposta dalla Microchip nei propri *datasheet* presenta tre piccole lacune, per cui l'autore, anche in questo caso, si è preso la licenza di una piccola modifica alla circuiteria ufficiale proposta dalla casa costruttrice.

Detta modifica, marginale, verrà comunque puntualmente evidenziata, in modo che il lettore possa autonomamente scegliere quella di sua preferenza.

Nella fig. 4.36 è evidenziata l'architettura dei pin di I/O del PORTE.

Figura 4.36: Architettura del PORTE

La modifica introdotta rispetto al circuito proposto dalla Microchip nei propri fogli tecnici è stata direttamente evidenziata in rosso. Il collegamento fra pin di I/O e convertitore A/D, risulta essere infatti assente (cfr.DS30292C [5]), come pure il collegamento verso i tre segnali di *handshake* del PSP.

Vi è inoltre un terzo elemento del circuito che non illustra alla perfezione l'architettura del PORTE: il buffer di ingresso è di tipo *Schmitt Trigger* se il pin è programmato come I/O digitale e di tipo TTL se il pin è programmato come pin ausiliario del PSP.

Ciò non appare in maniera evidente dal circuito di fig.4.36. Comunque, molti dubbi troveranno risposta dopo lo studio attento delle periferiche correlate al PORTE.

Infine, come tutti i port dei PIC16F87x, anche gli ingressi del PORTE sono protetti mediante diodi di *clamping* dalle sovratensioni positive e negative.

4.1.5.2 La programmazione del PORTE

Anche l'inizializzazione del PORTE è piuttosto semplice, essendo coinvolti solamente il *Parallel Slave Port* ed il convertitore analogico-digitale.

Sarà quindi sufficiente disabilitare le suddette due periferiche per poter utilizzare i tre pin del PORTE come pin di I/O digitale.

In fig. 4.37 sono evidenziati i registri che influenzano la programmazione del PORTE.

Address	Name	bit 7	bit 6	bit 5	bit 4	bit 3	bit 2	bit 1	bit 0	Value on POR, BOR	Value on all other RESETS
09h, 109h	PORTE	/	/	/	/	/	RE2	RE1	RE0	xxxx xxxx	uuu uuuu
89h, 189h	TRISE	IBF	OBF	IBOV	PSPMODE	/	PORTE Data Direction bits			0000 -111	0000 -111
9Fh	ADCON1	ADFM	/	/	/	PCFG3	PCFG2	PCFG1	PCFG0	--0- 0000	--0- 0000

Legend: x = unknown, u = unchanged, / = unimplemented locations read as 0
Shaded cells are not used by PORTE

Figura 4.37: Registri coinvolti nella programmazione del PORTE

Il programma di inizializzazione del PORTE diventa quindi il seguente:

Listing 4.5: Inizializzazione PORTE

```
banksel PORTE   ;Select Bank 0
clrf    PORTE   ;Initialize PORTE by clearing
                ;output data latches
banksel TRISE
movlw   0x06    ;Configure all pins
movwf   ADCON1  ;as digital inputs
movlw   0x03    ;Disables PSP mode and
movwf   TRISE   ;set RE<1:0> as inputs
                ;and RE<3> as output.
```

4.1.6 Considerazioni aggiuntive sui port di I/O

Le prossime due sezioni andrebbero entrambe evidenziate con il segnale di pericolo, dato che vi sono alcune considerazioni aggiuntive da fare legate al funzionamento dei port di I/O che, se sottovalutate, potrebbero diventare fonte di incomprensibili errori, piuttosto difficili da individuare e correggere se si è sprovvisti del necessario *know-how* tecnologico. Tali considerazioni riguardano sostanzialmente due aspetti particolarmente importanti: le cosidette istruzioni *read-modify-write* e le operazioni di I/O in sequenza.

4.1.6.1 Le istruzioni di *read-modify-write*

Tutte le istruzioni che coinvolgono una scrittura su periferica o in memoria sono in realtà istruzioni che *prima* eseguono la lettura del dato destinazione, lo modificano e *dopo* eseguono la scrittura. Queste istruzioni sono dette, appunto di *read-modify-write*.

Dette istruzioni sono potenzialmente pericolose quando applicate ai port di I/O bidirezionali. Si supponga, ad esempio, che il PORTB sia programmato in modo tale che i 4 bit più significativi siano inizialmente programmati come ingressi e che i 4 bit meno significativi siano programmati come uscite. Si supponga altresì che i due bit più significativi siano mantenuti a livello logico 1 da due *pull up* esterni. La situazione circuitale ed i relativi (esemplificativi) stati logici è illustrata in fig. 4.38.

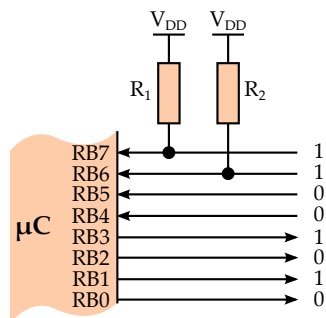



Figura 4.38: Circuito con I/O bidirezionali

Si supponga ora che vi siano due operazioni di *read-modify-write* in sequenza, come evidenziato dal sottostante codice:

Listing 4.6: Operazioni di *read-modify-write*

```
;Initial PORT settings: PORTB<7:4> Inputs
;PORTB<3:0> Outputs
;PORTB<7:6> have external pull-ups and aren't connected to other
;circuitry
;
;                       PORT latch      PORT pins
;                       ----------      ---------
1.      banksel PORTB
2.      bcf PORTB,7     ;0100 1010      1100 1010
3.      bcf PORTB,6     ;1000 1010      1100 1010
4.      banksel TRISB
5.      bcf TRISB,7     ;1000 1010      1100 1010
6.      bcf TRISB,6     ;1000 1010      1000 1010
```

L'istruzione 1 seleziona il banco relativo al PORTB e l'istruzione 2 azzera il bit 7 del *PORTB Latch*. Quello che realmente avviene durante l'esecuzione dell'istruzione 2 è quanto segue:

1. *tutto il PORTB* viene letto. I quattro bit meno significativi sono letti dal *PORTB Latch*, mentre i quattro bit più significativi, essendo ingressi, sono letti dall'*Input Latch* (vedi sez.4.1.2.3);
2. viene operata la modifica sul byte letto da parte dell'ALU;
3. viene memorizzato il nuovo valore sul PORTB (non sui pin d'uscita). In tal modo vengono effettivamente aggiornati i quattro bit meno significativi (essendo uscite e, quindi, gestite direttamente dal *PORTB Latch*), ma non i quattro bit più significativi che sono ingressi e non possono essere alterati dal *PORTB Latch*.

Durante l'esecuzione dell'istruzione 3, però, succede un guaio.

I tre passi precedentemente descritti vengono rieseguiti e durante il primo di detti passi viene riletto l'intero PORTB. Siccome, però, il bit 7 viene letto dall'*Input Latch* e non dal *PORTB Latch*, durante la lettura esso viene letto come stato logico 1 e non come stato logico 0, come un ingenuo programmatore, dopo l'esecuzione dell'istruzione 2, potrebbe aspettarsi.

Dopo l'esecuzione dell'istruzione 3 il *PORTB Latch* contiene, quindi, il valore 1000 1010 e non 0000 1010 come il programmatore (ingenuo) immagina.

Quando nelle successive tre istruzioni i bit 7 e 6 vengono programmati come uscita, il risultato non sarà quello atteso: sui pin del PORTB si avrà il valore 1000 1010.

Molta attenzione si dovrà prestare, quindi, nell'uso delle istruzioni di *read-modify-write* in un contesto di variabilità delle linee di I/O come quello appena illustrato.

4.1.6.2 Operazioni di I/O in sequenza

Anche le operazioni di scrittura e lettura operate in sequenza sullo stesso port possono creare qualche dispiacere. Si supponga il seguente codice:

Listing 4.7: Operazioni successive sullo stesso port

```
movwf   PORTB   ;Scrittura PORTB
movf    PORTB,W ;Lettura PORTB
nop
nop
```

Con riferimento alla figura sottostante si può esaminare in maggior dettaglio i momenti in cui avvengono la scrittura e la lettura.

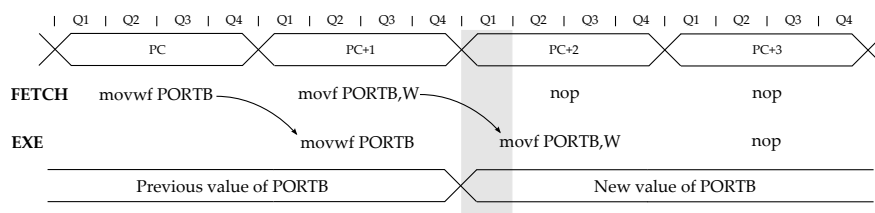



Figura 4.39: Operazioni successive sullo stesso port

Durante il ciclo macchina PC+1 viene effettuato il *fetch* della istruzione `movf PORTB,W` ed eseguita l'istruzione `movwf PORTB`.

La scrittura del PORTB ha effetto allo scadere del quarto ciclo di clock del ciclo macchina PC+1. L'istruzione `movf PORTB,W` viene eseguitamentre durante il ciclo macchina PC+2. Sfortunatamente, però, il campionamento del dato contenuto nel PORTB avviene allo scadere del primo ciclo di clock del corrente ciclo macchina. Ciò significa che fra la scrittura e la lettura vi è solamente un tempo pari ad un ciclo di clock (evidenziato in grigio nella figura).

Se la frequenza di lavoro del micro è alta e/o se il port "vede" una capacità parassita consistente, è possibile che al termine del ciclo di clock Q1 il dato non sia completamente pronto e la lettura viene effettuata con errore.

E' quindi preferibile aggiungere un'istruzione `nop` fra la scrittura e la successiva lettura, in modo da dare al dato contenuto nel PORTB di stabilizzarsi, come proposto nel seguente listato:

Listing 4.8: Operazioni successive corrette

```
movwf   PORTB   ;Scrittura PORTB
nop             ;Assestamento del dato
movf    PORTB,W ;Lettura PORTB
nop
nop
```

La situazione appena descritta è tipica di scritture/letture su collegamenti (solitamente esterni alla scheda) di una certa lunghezza (qualche metro).

4.2 Il modulo USART

USART significa *Universal Synchronous Asynchronous Receiver Transmitter*. Si tratta di una periferica di comunicazione seriale asincrona/sincrona utilizzabile per comunicazioni *full duplex* di tipo RS232C, RS422, RS485, ecc.

Tale periferica occupa solamente due pin del PORTC, precisamente i pin RC7 e RC6, attribuiti rispettivamente alla linea di ricezione e alla linea di trasmissione del modulo. Nel caso in cui venga utilizzata una comunicazione non *peer to peer* sarà necessario aggiungere una terza linea avente funzione di direzione, prendendo un qualsiasi bit a disposizione da un qualsiasi port.

La periferica può essere programmata in uno dei seguenti tre modi:

- comunicazione asincrona *full duplex*;
- comunicazione sincrona in modalita Master *half duplex*;
- comunicazione sincrona in modalita Slave *half duplex*.

Prima di addentrarci nello studio dettagliato dell'USART si preferisce, però, consigliare una veloce lettura dei concetti fondamentali delle comunicazioni asincrone. A tal proposito si rimanda al capitolo **??** (con particolare riferimento alle prime 5 sezioni) che contiene una breve e fondamentale introduzione alle comunicazioni seriali.

4.2.1 Il *Baud Rate Generator*

Una parte assolutamente fondamentale di un USART è il *Baud Rate Generator*, avente il compito di scandire il tempo di ciascun modulo USART indipendente.

La comunicazione asincrona si fonda proprio sul principio che i due dispositivi comunicanti siano dotati ciascuno di un orologio molto preciso da sincronizzare all'inizio della comunicazione. Tale orologio è, appunto, il *Baud Rate Generator* (BRG).

Esso fornisce un clock alla esatta frequenza scelta per comunicare serialmente. Storicamente, le seguenti erano frequenze normalmente usate per le comunicazioni asincrone:

300, 600, 1200, 2400, 4800, 9600, 19200, 28800, 38400, 57600, 115200,

dove l'unità di misura dei sopra elencati valori è il Hertz (Hz).

Oggidì le frequenze possono essere, anche in campo industriale, molto più alte, oltre i 10MHz. Il PIC16F877, però, si ferma a 1.25MHz.

L'esatta frequenza di clock generata dal BRG dipende sostanzialmente da tre parametri:

1. dalla frequenza di oscillazione Fosc fornita al micro;
2. dal valore impostato nel registro a 8 bit SPBRG;
3. dal valore del bit BRGH.

Utilizzando i suddetti tre parametri è possibile usare la seguente formula per calcolare il *baud rate* effettivo di comunicazione:

Communication	BRGH = 0 (Low speed)	BRGH = 1 (High speed)
Asynchronous	br = Fosc/(64(SPBRG+1))	br = Fosc/(16(SPBRG+1))
Synchronous	br = Fosc/(4(SPBRG+1))	N.A.

Tabella 4.7: Calcolo del *baud rate* effettivo

La fig. 4.40 riassume brevemente i possibili *baud rate* in funzione dei tre parametri, indicando l'errore sul *baud rate* ideale.

Baud rate for asynchronous mode (BRGH = 0)

Baud Rate (K)	Fosc = 20MHz			Fosc = 16MHz			Fosc = 10MHz			Fosc = 4MHz			Fosc = 3.6864MHz		
	KBaud	% Error	SPBRG value	KBaud	% Error	SPBRG value	KBaud	% Error	SPBRG value	KBaud	% Error	SPBRG value	KBaud	% Error	SPBRG value
0.3	/	/	/	/	/	/	/	/	/	0.300	0	207	0.3	0	191
1.2	1.221	1.75	255	1.202	0.17	207	1.202	0.17	129	1.202	0.17	51	1.2	0	47
2.4	2.404	0.17	129	2.404	0.17	103	2.404	0.17	64	2.404	0.17	25	2.4	0	23
9.6	9.766	1.73	31	9.615	0.16	25	9.766	1.73	15	8.929	6.99	6	9.6	0	5
19.2	19.531	1.72	15	19.231	0.16	12	19.531	1.72	7	20.833	8.51	2	19.2	0	2
28.8	31.250	8.51	9	27.778	3.55	8	31.250	8.51	4	31.250	8.51	1	28.8	0	1
33.6	34.722	3.34	8	35.714	6.29	6	31.250	6.99	4	/	/	/	/	/	/
57.6	62.500	8.51	4	62.500	8.51	3	52.083	9.58	2	62500	8.51	0	57.6	0	0
HIGH	1.221	/	255	0.977	/	255	0.610	/	255	0.244	/	255	0.225	/	255
LOW	312.500	/	0	250.000	/	0	156.250	/	0	62.500	/	0	57.6	/	0

Baud rate for asynchronous mode (BRGH = 1)

Baud Rate (K)	Fosc = 20MHz			Fosc = 16MHz			Fosc = 10MHz			Fosc = 4MHz			Fosc = 3.6864MHz		
	KBaud	% Error	SPBRG value	KBaud	% Error	SPBRG value	KBaud	% Error	SPBRG value	KBaud	% Error	SPBRG value	KBaud	% Error	SPBRG value
0.3	/	/	/	/	/	/	/	/	/	/	/	/	/	/	/
1.2	/	/	/	/	/	/	/	/	/	1.202	0.17	207	1.2	0	191
2.4	/	/	/	/	/	/	2.441	1.71	255	2.404	0.17	103	2.4	0	95
9.6	9.615	0.16	129	9.615	0.16	103	9.615	0.16	64	9.615	0.16	25	9.6	0	23
19.2	19.231	0.16	64	19.231	0.16	51	19.531	1.72	31	19.231	0.16	12	19.2	0	11
28.8	29.070	0.94	42	29.412	2.13	33	28.409	1.36	21	27.798	3.55	8	28.8	0	7
33.6	33.784	0.55	36	33.333	0.79	29	32.895	2.10	18	35.714	6.29	6	32.9	2.04	6
57.6	59.524	3.34	20	58.824	2.13	16	56.818	1.36	10	62500	8.51	3	57.6	0	3
HIGH	4.883	/	255	3.906	/	255	2.441	/	255	0.977	/	255	0.9	/	255
LOW	1250.00	/	0	1000.00	/	0	625.000	/	0	250.000	/	0	230.4	/	0

Figura 4.40: Tabella dei *baud rate*

La tabella di fig. 4.40 utilizza valori comuni di frequenza di oscillazione. Il progettista è naturalmente libero di scegliere il valore di frequenza che preferisce, anche al di fuori da quelli proposti. In termini generali, frequenze alte sono da preferirsi, a meno che il micro non sia alimentato a batteria, nel qual caso vale l'inverso.

Può sembrare strana la frequenza di 3.6864MHz. E' invece, piuttosto usata proprio perché garantisce errori relativi percentuali pari a 0 su tutti i *baud rate* standard.

Microchip fornisce anche un esempio di calcolo del valore di SPBRG e dell'errore relativo percentuale, supponendo una frequenza di clock di 16MHz ed un *baud rate* desiderato di 9600:

$$Desidered\ Baud\ Rate = Fosc/(64(SPBRG + 1)) \tag{4.2}$$

$$9600 = 16000000/(64(SPBRG + 1))$$

$$SPBRG = \lfloor 25.042 \rfloor = 25 \tag{4.3}$$

$$Calculated\ Baud\ Rate = 16000000/(64(25 + 1)) \tag{4.4}$$

$$= 9615$$

$$Error = \frac{Calculated\ Baud\ Rate - Desired\ Baud\ Rate}{Desired\ Baud\ Rate} \tag{4.5}$$

$$= (9615 - 9600)/9600$$

$$= 0.16\%$$

Rimane da stabilire quali errori relativi percentuali sui valori di *baud rate* sono accettabili e quali no. Lo studente non metodico potrebbe sostenere che un errore dello 0.16% è sicuramente accettabile, mentre un errore del 9.58% non lo è.

Si può sicuramente essere d'accordo con una siffatta affermazione, ma è sempre meglio fare qualche calcolo prima di sbilanciarsi. Purtroppo non è possibile eseguirlo finché non si conosce dettagliatamente il metodo di campionamento del PIC16F877. Anzi, due sono gli aspetti da studiare correlati al campionamento della linea di ricezione seriale: la sincronizzazione del segnale ed il successivo campionamento dei bit. Una introduzione semplificata alla sincronizzazione del bit di Start è fornita nella prissima sezione.

4.2.2 Una sincronizzazione semplificata

Normalmente il segnale di ricezione, in stato di *idle*, è a livello logico 1[11]. La sincronizzazione viene operata dalla stazione ricevente nel momento in cui arriva il segnale di Start. Più precisamente, il ricevitore campiona continuamente la linea di ricezione cercando di agganciare il fronte di discesa indicante la presenza del bit di Start.

[11]Non si devono confondere i livelli di tensione *in linea* da quelli sui pin dell'USART. Le tensioni di cui si parla nelle presenti sezioni sono sempre riferite ai pin della periferica di comunicazione seriale.

La bontà della sincronizzazione iniziale dipende quindi dalla frequenza di campionamento della linea. Una esemplificazione grafica di quanto esposto è data in fig. 4.41, dove il clock di campionamento del trasmettitore ha una frequenza leggermente più alta di quella del ricevitore e dove i due segnali sono asincroni fra loro.

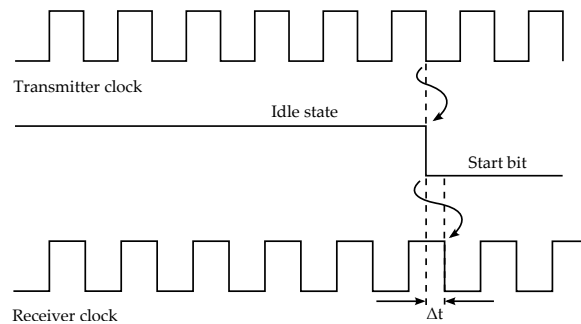



Figura 4.41: Sincronizzazione dello *start bit*

Il bit di Start viene prodotto dal trasmettitore in maniera sincrona con il proprio clock, ma ricevuto in maniera asincrona con un ritardo Δt massimo pari al periodo di clock del ricevitore.

La situazione rappresentata in fig. 4.41 è però alquanto semplificata, dato che la situazione reale è leggermente più complessa.

4.2.3 La sincronizzazione reale

In realtà il *Baud Rate Generator* produce due forme d'onda: un *Baud CLK* alla frequenza del *baud rate* impostato ed un segnale a frequenza 16 volte maggiore, come evidenziato in fig. 4.42.

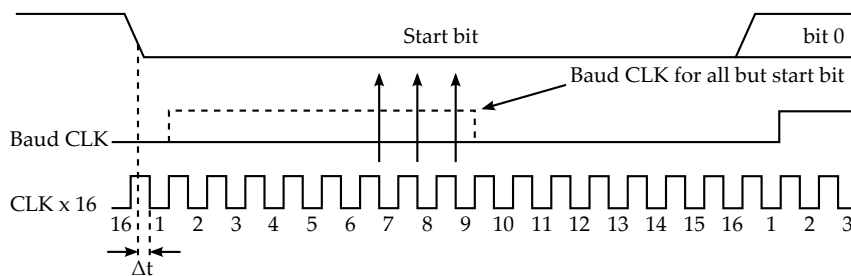



Figura 4.42: Sincronizzazione reale

Quest'ultimo segnale è utilizzato dall'USART del ricevitore per campionare in maniera asincrona la linea di ricezione RX. Il fronte di discesa del bit di Start verrà agganciato con un certo ritardo variabile Δt il cui valor massimo coincide con il periodo del clock.

Una volta agganciato il clock, l'USART conta 16 impulsi e lancia un secondo segnale di clock, questa volta alla frequenza del *baud rate*. Si noti, quindi, che per tutto il periodo del bit di Start tale segnale è a livello logico 0.

Il campionamento del bit di Start viene effettuato mediante tre letture consecutive della linea RX: rispettivamente sul fronte di discesa del settimo, ottavo e nono bit del clock ad alta frequenza. Un siffatto rilevamento permette di rilevare eventuali *glitches* senza confonderli erroneamente con lo stato del bit.

Il campionamento del bit di Start non avviene quindi esattamente nel mezzo del periodo di bit, come sarebbe corretto, ma con un errore Δt dato dal non perfetto aggancio del fronte di discesa del bit di Start. Tale imperfetto aggancio forma l'errore di sincronizzazione.

4.2.4 Il campionamento dei bit successivi allo Start

Il campionamento dei bit successivi allo Start segue la stessa identica procedura del campionamento dello Start stesso. Ciò che, invece, è interessante valutare è l'ulteriore errore che si compie durante il campionamento.

Si è visto che la sincronizzazione è viziata da un aggancio non perfetto del fronte di discesa dello Start. Tale errore è compiuto *una tantum* all'inizio di ogni carattere. per i successivi 10 bit seguenti lo Start, si suppone che il trasmettitore ed il ricevitore siano sincronizzati.

Ciò non è, però, completamente vero, perché i due BRG generano due frequenze molto simili ma sicuramente non perfettamente uguali. All'errore di sincronizzazione si deve pertanto sommare anche il *drift* di frequenza che potrebbe diventare determinante.

Se l'errore relativo percentuale sul *baud rate* ideale è consistente il campionamento può velocemente derivare. In fig. 4.43 è rappresentato una tale situazione[12].

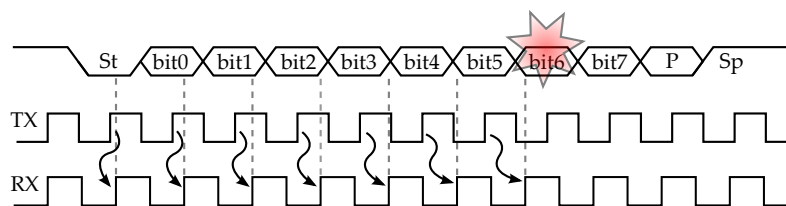



Figura 4.43: Campionamento errato per *drift* di frequenza

Si suppone che in fig. 4.43 il clock generato dal trasmettitore sia idealmente dello stesso valore del clock standard (ad esempio, 57600 bps) e che quello del ricevitore sia più lento del 9.58%. Per semplicità si suppone anche che che non vi sia errore di sincronizzazione.

Il *pattern* di bit generato dal trasmettitore, perfettamente sincrono con il clock del trasmettitore, viene campionato a metà del bit[13] dal ricevitore. Essendo, però, la frequenza di campionamento più lenta,il quarto ed il quinto

[12]Si è cercato di mantenere la proporzione grafica, per cui l'effetto prodotto è proprio quello di una frequenza di campionamento in lettura del 9.58% più lenta.

[13]Si è visto che in realtà i campionamenti sono tre, effettuati *circa* a metà del periodo di bit.

campionamento sono già dubbi ed il sesto è definitivamente effettuato nel bit errato: invece di campionare il bit 5, viene campionato il bit 6, con evidente errore.

4.2.5 Calcolo dell'errore percentuale massimo

Ancora non si è data una risposta su quali errori relativi percentuali siano accettabili e quali no. La questione era stata sollevata nella sezione 4.2.1.

Si è visto che le tecniche di sincronizzazione e campionamento sono diverse e dipendono dal valore impostato nel bit `BRGH`. Nella presente sezione si eseguirà il calcolo dell'errore relativo percentuale massimo ammissibile in una comunicazione seriale supponendo `BRGH = 0` e una carattere di 10 bit: 1 bit di Start, uno di Stop e 8 bit di dato, quindi senza Parità. Si dovranno esaminare due casi distinti: 1) la frequenza di campionamento del ricevitore è minore di quella del trasmettitore; 2) la frequenza di campionamento del ricevitore è maggiore di quella del trasmettitore. Si ipotizzerà, per semplicità, che il trasmettitore sia ideale, trasmetta, cioè, con errore relativo percentuale pari a 0.

4.2.5.1 Frequenza di campionamento minore

Sia t_{bit} il periodo di bit scelto per la comunicazione seriale. La trasmissione del carattere durerà quindi $10t_{bit}$. L'errore di sincronizzazione assoluto può valere al massimo $t_{bit}/16$ al quale va sommato l'errore assoluto massimo d_{max} dovuto al *drift* di frequenza che si vuole calcolare. Siccome l'errore assoluto massimo ammesso durante il campionamento è di $(8.5/16)t_{bit}$[14] si ha

$$d_{max} = \frac{8.5}{16}t_{bit} - \frac{t_{bit}}{16} = \frac{7.5}{16}t_{bit} = 0.46875t_{bit} \tag{4.6}$$

Siccome l'errore assoluto massimo di *drift* è accumulato su 10 bit, sul singolo bit si ha:

$$db_{max} = \frac{7.5}{160}t_{bit} = 0.046875t_{bit} \tag{4.7}$$

da cui si calcola l'errore relativo percentuale massimo ammissibile:

$$db\%_{max} = \frac{0.046875t_{bit}}{t_{bit}}100 = 4.6875\% \tag{4.8}$$

Applicando al risultato finale un coefficiente di sicurezza pari a 1.5, si ottiene un errore relativo percentuale massimo $db\%_{max}$ pari al 3.125%.

[14]Con riferimento alla fig. 4.42 a pagina 125, si nota che l'aggancio del fronte di discesa del bit di Start avviene sul fronte di discesa del segnale CLKx16, mentre il segnale Baud CLK è sincronizzato sul fronte di salita, da cui il mezzo bit in più.

4.2.5.2 Frequenza di campionamento maggiore

Se, invece, la frequenza di campionamento del ricevitore è maggiore di quella del trasmettitore è necessario fare altre considerazioni.

Nel caso precedente, se la frequenza del ricevitore era minore di quella del trasmettitore, c'era il rischio di campionare erroneamente il bit successivo a quello che effettivamente avrebbe dovuto essere campionato. E' ciò che effettivamente è successo nell'esempio illustrato in fig. 4.43.

Se, invece, la frequenza del ricevitore è maggiore di quella del trasmettutore si corre il rischio di campionare due volte lo stesso bit, come esemplificato graficamente in fig. 4.44.

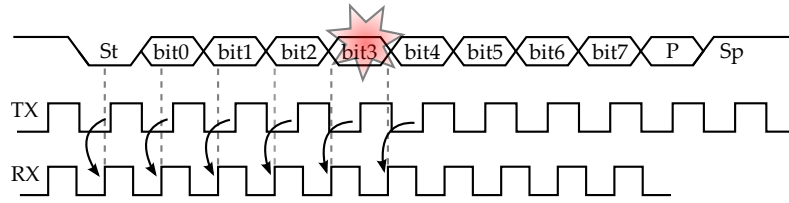



Figura 4.44: Campionamento doppio del bit 3

Per poter valutare l'errore relativo percentuale massimo ammesso, si deve supporre che *non vi sia ritardo nell'aggancio* del fronte di discesa del bit di Start. Ciò significa errore di sincronizzazione pari a 0.

In tal caso il *drift* massimo ammissibile è dato da:

$$d_{max} = \frac{6}{16} t_{bit} = 0.375 t_{bit} \tag{4.9}$$

Siccome anche in questo caso l'errore assoluto massimo di *drift* è accumulato su 10 bit, sul singolo bit si ha:

$$db_{max} = \frac{6}{160} t_{bit} = 0.0375 t_{bit} \tag{4.10}$$

da cui si calcola l'errore relativo percentuale massimo ammissibile:

$$db\%_{max} = \frac{0.0375 t_{bit}}{t_{bit}} 100 = 3.75\% \tag{4.11}$$

Applicando al risultato finale un coefficiente di sicurezza pari a 1.5, si ottiene un errore relativo percentuale massimo $db\%_{max}$ pari al 2.5%.

Non volendo distinguere i due casi, ma volendo essere certi di scegliere sempre correttamente il *baud rate* di comunicazione seriale, si deve scegliere il *worst case*. Si consiglia pertanto di orientarsi su valori di errore relativo percentuale inferiore al 2.5%.

4.2.6 Il registro TXSTA

Due sono i registri fondamentali che riguardano la programmazione del modulo USART del PIC16F877: il registro di trasmissione TXSTA ed il registro di ricezione RCSTA. Attraverso tali registri si programma gran parte della periferica di comunicazione seriale. La presente sezione esamina il primo dei due registri.

La struttura del registro TXSTA è la seguente:

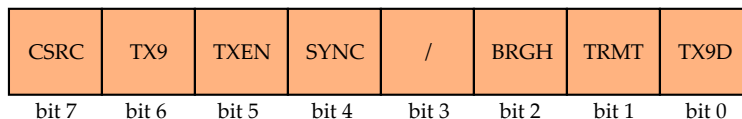



Figura 4.45: Il registro TXSTA

I singoli bit, partendo dal più significativo, assumono il seguente significato:

- CSRC: Clock Source Select bit
  Asynchronous mode:
    Don't care
  Synchronous mode:
    1 = Master mode (clock generated internally from BRG)
    0 = Slave mode (clock from external source)
- TX9: 9-bit Transmit Enable bit
    1 = Selects 9-bit transmission
    0 = Selects 8-bit transmission
- TXEN: Transmit Enable bit
    1 = Transmit enabled
    0 = Transmit disabled
  Note: SREN/CREN overrides TXEN in SYNC mode.
- SYNC: USART Mode Select bit
    1 = Synchronous mode
    0 = Asynchronous mode
- Unimplemented: Read as '0'
- BRGH: High Baud Rate Select bit
  Asynchronous mode:
    1 = High speed
    0 = Low speed
  Synchronous mode:
    Unused in this mode
- TRMT: Transmit Shift Register Status bit
    1 = TSR empty
    0 = TSR full
- TX9D: 9th bit of Transmit Data, can be parity bit

IL bit CSRC indica la sorgente del clock quando la periferica viene programmata in modalità sincrona: se posto a 1, significa che la periferica, quando è programmata in modalità sincrona, funziona da Master e quindi ha il compito di fornire il clock allo Slave attraverso il pin RC6. La sorgente di clock del Master diventa quindi il BRG.

Se tale bit è posto a 0, significa che la periferica, quando è programmata in modalità sincrona, funziona da Slave e quindi la sorgente del clock deve essere fornita dal Master sul proprio pin RC6.

Se la periferica è programmata in modalità asincrona, detto bit perde significato.

L'USART può essere programmato per trasmettere 8 oppure 9 bit di dato. Il non bit *solitamente* assume il significato di bit di parità. Si deve però sottolineare che è compito del programmatore calcolare il bit di parità, dato che l'USART non vi provvede automaticamente. In tal caso il non bit può essere usato per memorizzarvi la parità calcolata.

Il bit TX9, se posto a 1, abilita l'USART alla trasmissione del nono bit.

Il bit TXEN abilita il trasmettitore. Finché tale bit rimane a 0 logico, il trasmettitore non inizia la trasmissione. Si noti che il bit TXEN viene sovrascritto dai bit SREN e CREN. Vedi a tal proposito il registro RCSTA nella prossima sezione.

Il bit SYNC è responsabile della modalità di trasmissione: se posto a 1 logico, abilita la periferica alla trasmissione sincrona; se posto a 0 logico abilita la periferica alla trasmissione asincrona.

Il bit BRGH assume un senso solamente se la periferica è stata programmata in modalità asincrona, In tal caso se il bit BRGH è posto a 1 viene abilitato il *high speed mode* ed il *Baud Rate Generator* viene messo nelle condizioni migliori per generate dei *baud rate* elevati. Se il bit BRGH è posto a 0 logico, si abilita il *low speed mode*, indicato per la generazione di *baud rate* a frequenza più bassa.

Se la periferica è programmata in modalità sincrona detto bit non ha significato.

Il bit TMRT è un bit a sola lettura e non può, quindi, essere scritto dal programmatore. Esso indica lo stato del registro di trasmissione seriale: se il bit TMRT vale 0 il registro è già caricato con il dato da trasmettere; se il bit vale 1 il registro è vuoto e può essere caricato per effettuare la prossima trasmissione.

Il bit TX9D è il non bit che viene trasmesso se è stata abilitata la trasmissione a nove bit. E' compito del programmatore scriverci un valore adeguato. Solitamente il valore scritto rappresenta la parità pari o dispari del dato a 8 bit.

Per ulteriori dettagli sulla parità e sul suo uso si veda la sezione **??**.

4.2.7 Il registro RCSTA

Si è visto nella sezione precedente che il registro TXSTA ha il compito di gestire la trasmissione sincrona/asincrona effettuata dall'USAT. Analogamente il registro RCSTA ha il compito di gestire la ricezione sia essa in modalità sincrona che asincrona.

Inoltre, il registro RCSTA è responsabile dell'abilitazione dell'USART o meno e, quindi, indirettamente anche della destinazione d'uso dei pin RC6 e RC7.

Infatti, se la periferica di comunicazione seriale viene disabilitata, automaticamente i pin RC6 e RC7 possono essere utilizzati dal programmatore come pin di I/O generici, mentre se la periferica viene abilitata, detti pin hanno un ruolo ben preciso definito dai registri TXSTA e RCSTA.

La struttura del registro RCSTA è la seguente:

Figura 4.46: Il registro RCSTA

I singoli bit assumono il seguente significato:

- SPEN: Serial Port Enable bit
  1 = Serial port enabled (RC7 and RC6 are serial port pins)
  0 = Serial port disabled
- RX9: 9-bit Receive Enable bit
  1 = Selects 9-bit reception
  0 = Selects 8-bit reception
- SREN: Single Receive Enable bit
  Asynchronous mode:
  Don't care
  Synchronous mode - master:
  1 = Enables single receive
  0 = Disables single receive
  This bit is cleared after reception is complete.
  Synchronous mode - slave:
  Don't care
- CREN: Continuous Receive Enable bit
  Asynchronous mode:
  1 = Enables continuous receive
  0 = Disables continuous receive
  Synchronous mode:
  1 = Enables continuous receive until enable bit CREN is cleared (CREN overrides SREN)
  0 = Disables continuous receive
- ADDEN: Address Detect Enable bit
  Asynchronous mode 9-bit (RX9 = 1):
  1 = Enables address detection, enables interrupt and load of the receive buffer when RSR<8> is set
  0 = Disables address detection, all bytes are received, and ninth bit can be used as parity bit
- FERR: Framing Error bit
  1 = Framing error (can be updated by reading RCREG register) and receive next valid byte)
  0 = No framing error
- OERR: Overrun Error bit
  1 = Overrun error (can be cleared by clearing bit CREN)
  0 = No overrun error

- RX9D: 9th bit of Received Data (can be parity bit, but must be calculated by user firmware)

Il bit SPEN è il bit che abilita o meno la periferica di comunicazione seriale. Se il bit è posto a 0 logico i pin RC6 e RC7 possono essere usati dal programmatore come pin di I/O generici. Se il bit è posto a 1 logico la periferica di comunicazione seriale viene abilitata e i pin RC6 e RC7 sono dedicati rispettivamente alla ricezione seriale asincrona o alla linea dati sincrona e alla trasmissione seriale asincrona o alla linea di clock sicrona.

Il bit RX9 assume lo stesso significato del bit TX9 del registro TXSTA: seleziona, se posto a 1, la ricezione a nove bit oppure, se posto a 0, la ricezione a otto bit.

Il bit SREN ha significato solamente se la periferica è programmata in modalità sincrona. Se l'USART è programmato come Master detto bit abilita la ricezione del singolo carattere. A ricezione completata, il bit viene automaticamente azzerato. E' compito del programmatore risettarlo per la prossima ricezione singola. In modalità Slave il bit non ha significato.

Il bit CREN abilita, in modalità sincrona, la ricezione continua, sovrascrivendo il bit SREN. Il bit assume lo stesso significato anche in modalità asincrona. Si noti che, in modalità sincrona, i bit CREN e SREN sovrascrivono il bit TXEN del registro TXSTA.

Il bit ADDEN abilita, se posto a 1, l'*address detecting*. Si tratta di una funzionalità offerta dal PIC16F87x per un veloce riconoscimento dell'indirizzo di rete di un dispositivo di comunicazione. In tal caso il nono bit viene utilizzato per la composizione dell'indirizzo di rete (formato proprio da nove bit) e non può essere utilizzato per la memorizzazione del bit di parità. Se si desidera sfruttare questa funzionalità del micro è necessario porre `RX9 = 1`.

Il bit FERR è un bit a sola lettura e identifica, se settato, l'errore di *framing* in fase di comunicazione. Tale errore è quasi sempre indice di problemi al *baud rate*: o eccessivamente impreciso e addirittura errato. Se il carattere ricevuto è invece privo di errore di *framing* il bit è posto a 0.

Il bit di OERR identifica l'errore di *overrun*. Si tratta di un bit a sola lettura, che può, però, essere azzerato ponendo a 0 logico il bit di CREN. L'errore di *overrun* si manifesta quando vengono ricevuti 3 caratteri consecutivi senza che nessuno dei tre sia effettivamente stato letto.

Il bit RX9D assume lo stesso significato del bit TX9D. Esso rappresenta il dato riferito al nono bit. Mentre il TX9D è il dato che viene scritto dal programmatore e trasmesso, il bit RX9D è il dato ricevuto e che il programmatore deve leggere. Si tratta, quindi, di un bit a sola lettura. Se esso rappresenta la parità pari o dispari della comunicazione, è compito del programmatore calcolare la parità e verificarla con quella ricevuta.

4.2.8 La programmazione del *Baud Rate Generator*

Alla luce del procedimento di calcolo e verifica illustrato nelle sezioni precedenti e della struttura dei due registri fondamentali della periferica di comunicazione seriale, si può riassumere quale debba essere il codice di inizializzazione del *Baud Rate Generator* e quali siano i registri coinvolti.

In fig. 4.47 sono evidenziati i registri che influenzano la programmazione del *Baud Rate Generator*.

Address	Name	bit 7	bit 6	bit 5	bit 4	bit 3	bit 2	bit 1	bit 0	Value on POR, BOR	Value on all other RESETS
98h	TXSTA	CSRC	TX9	TXEN	SYNC	/	BRGH	TRMT	TX9D	0000 -010	0000 -010
18h	RCSTA	SPEN	RX9	SREN	CREN	ADDEN	FERR	OERR	RX9D	0000 000x	0000 000x
99h	SPBRG	Baud Rate Generator Register								0000 0000	0000 0000
0Bh, 10Bh 8Bh, 18Bh	INTCON	GIE	PEIE	T0IE	INTE	RBIE	T0IF	INTF	RBIF	0000 000x	0000 000u
8Ch	PIE1	PSPIE	ADIE	RCIE	TXIE	SSPIE	CCP1IE	TMR2IE	TMR1IE	0000 0000	0000 0000
0Ch	PIR1	PSPIF	ADIF	RCIF	TXIF	SSPIF	CCP1IF	TMR2IF	TMR1IF	0000 0000	0000 0000

Legend: x = unknown, u = unchanged, / = unimplemented locations read as 0
Shaded cells are not used by the BRG

Figura 4.47: Registri coinvolti nella programmazione del BRG

Si supponga di voler inizializzare il BRG in modo da poter comunicare a 19200 bps partendo da una frequenza di clock del micro di 20MHz. Il codice potrebbe essere simile al seguente:

Listing 4.9: Inizializzazione del BRG

```
banksel TXSTA   ;Select TXSTA register's bank
bcf     TXSTA,SYNC      ;Select asynchronous mode
bcf     TXSTA,BRGH      ;Low speed baud rate
movlw   0x0F            ;Select 19200 bps
movwf   SPBRG
banksel RCSTA

;The following settings must be made when necessary
bsf     RCSTA,SPEN      ;USART Enable
bcf     PIR1,RCIR       ;Clear receive flag
bcf     PIR1,TXIR       ;Clear transmit flag
banksel PIE1
bsf     PIE1,RCIE       ;RX Interrupt Enable
bsf     PIE1,TXIE       ;TX Interrupt Enable
bsf     INTCON;PEIE     ;Peripheral Interrupt Enable
bsf     INTCON,GIE      ;General Interrupt Enable
```

4.2.9 La modalità asincrona

In modalità asincrona la periferica utilizza una codifica di linea di tipo NRZ (vedi sezione **??** a pagina ??) con 8 oppure 9 bit di dato un bit di Start ed un bit di Stop. Se si utilizza il formato a 8 bit di dato non è prevista la trasmissione della parità sul carattere. Se si desidera trasmettere il carattere con parità pari o dispari, si deve abilitare la comunicazione a nove bit (`RX9 = 1` e `TX9 = 1`) e rinunciare all'*address detecting* dell'indirizzo di rete.

Inoltre, è compito del programmatore calcolare la parità sul carattere prima di trasmetterla e verificarla in ricezione. La prossima sezione esamina questo aspetto della comunicazione.

4.2.9.1 L'uso della parità

Vi sono naturalmente molti modi per calcolare la parità di un carattere. Il modo più semplice consiste nell'effettuare 8 rotazioni attraverso il Carry dei bit del carattere e contare gli uni che vengono di volta in volta memorizzati nel Carry.

Microchip propone un metodo più semplice (cfr. PALACHERLA [2]), di cui si fornisce un semplice diagramma in fig. 4.48.

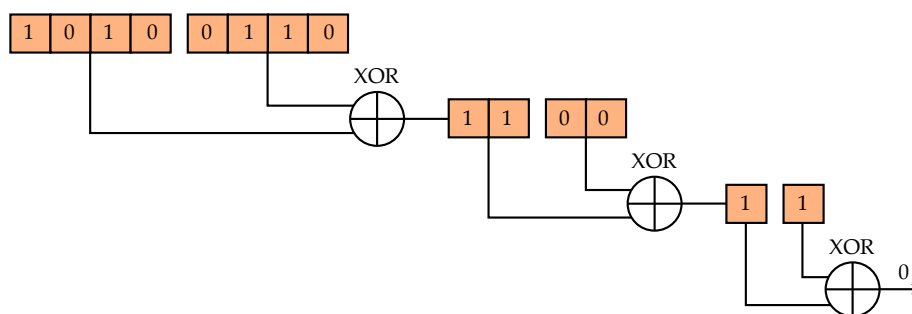



Figura 4.48: Diagramma del calcolo della parità

Si divide il carattere di cui calcolare la parità in due parti uguali e se ne esegue l'operazione logica di XOR. Il risultato lo si divide nuovamente in due parti uguali e si ricalcola la XOR e così via, fino ad ottenere un ultimo bit che rappresenta la *parità pari* del carattere originale. Se si intende comunicare con parità dispari, si dovrà invertire il risultato.

Nella stessa *application note* precedentemente citata viene offerta anche una *subroutine* di calcolo della parità. Essa è però piuttosto dispendiosa in termini di risorse: utilizza 3 registri, 13 istruzioni ed esegue 67 cicli macchina. Si tratta, quindi, di un codice piuttosto dispendioso.

Si coglie l'occasione per sottolineare l'importanza, in assembly, di scrivere codice non eccessivamente pesante, perché questo si traduce in spreco di risorse e cattiva efficienza, soprattutto in termini di velocità.

In quest'ottica, si propone ora un esempio di software per la generazione e verifica della parità che utilizza 1 registro, 12 istruzioni e 13 cicli macchina. Si suppone che il carattere di cui calcolare la parità sia posto in W.

Listing 4.10: Generazione/verifica della parità

```
GenPar: ;Generatore di parita'. Il risultato e' posto
        ;nel bit 0 di Parity
        banksel Parity  ;Selezione del banco
        movwf   Parity
        swapf   Parity  ;Dopo lo swap si ha:
                        ;W      = b7b6b5b4b3b2b1b0
                        ;Parity = b3b2b1b0b7b6b5b4
        xorwf   Parity,W;I due nibble di W contengono
                        ;entrambi il risultato della XOR
        movwf   Parity
        rrf     Parity
        rrf     Parity  ;Ora si puo' fare la XOR dei due
                        ;bit meno significativi fra W e
                        ;Parity
        xorwf   Parity,W
        movwf   Parity
        rrf     Parity  ;Ora si puo' fare la XOR del
                        ;bit meno significativo fra W e
                        ;Parity
        xorwf   Parity  ;La parita' e' nel bit meno
                        ;significativo di Parity
        return
```

Questa subroutine può essere usata sia in trasmissione che in ricezione. In fase di trasmissione dovrà essere copiato il bit `Parity<0>` nel bit TX9D del registro TXSTA, mentre in fase di ricezione il bit `Parity<0>` dovrà essere confrontato con il bit RX9D del registro RCSTA. Se la parità usata è dispari, ci si deve ricordare di invertire il bit `Parity<0>` della scrittura/lettura.

4.2.9.2 Il blocco di trasmissione

Il modulo di comunicazione asincrona è composto essenzialmente da

- il *Baud Rate Generator*;
- il circuito di campionamento;
- il blocco di trasmissione;
- il blocco di ricezione.

I primi due blocchi sono già stati esaminati. Nella presente sezione si analizzerà in dettaglio il blocco di trasmissione seriale asincrona.

Il trasmettitore è in grado di funzionare autonomamente ed in maniera trasparente al programmatore. Affinché la trasmissione seriale del carattere abbia inizio è sufficiente, dopo aver abilitato il modulo di comunicazione asincrona, caricare il carattere nel registro di trasmissione ed attendere l'interruzione di carattere inoltrato.

La trasmissione inizia con un bit di Start, cui seguono gli 8 bit del carattere a partire dal bit meno significativo (bit 0). Dopo gli 8 bit di carattere segue l'eventuale bit di parità e poi il bit di Stop. Se il micro viene posto in *stand by* con l'istruzione di `sleep`, la trasmissione si interrompe.

In fig. 4.49 è evidenziato il blocco di trasmissione asincrona del modulo USART. Direttamente interfacciato al Bus dei Dati si trova il registro TXREG, nel quale si carica il carattere di 8 bit da trasmettere. A caricamento avvenuto,

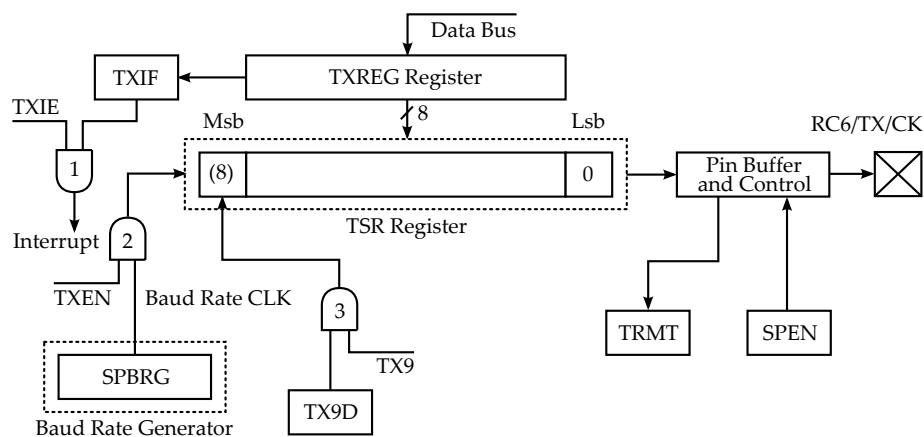



Figura 4.49: Blocco di trasmissione asincrona

se non vi sono trasmissioni in corso, ovvero se il bit di Stop è già stato trasmesso, nel ciclo macchina successivo al caricamento, il carattere viene trasferito nel registro di trasmissione seriale TSR (*Transmit Shift Register*) che scorre verso destra uno ad uno i singoli bit verso il pin di uscita RC6/TX/CK.

Si noti che il bit di Start e il bit di Stop non risiedono nel registro TSR, che è un registro di 8+1 bit, dove il nono bit dipende dallo stato del bit TX9. Lo Start e lo Stop vengono aggiunti nel buffer di uscita e controllo. Si noti che il registro TSR non è mappato in memoria, per non è accessibile in nessun modo al programmatore.

Correlati al buffer di uscita vi sono due bit: il bit di SPEN e il bit di TRMT. Il primo controllo il buffer di uscita, indicando esso l'abilitazione della periferica di comunicazione seriale. Il secondo bit ha il compito di monitorare lo stato dello *shift register* di trasmissione: se vale 1 significa che il registro si è svuotato e che la trasmissione del carattere è terminata[15].

Il clock del registro di scorrimento è fornito dalla porta AND 2 che, per poter porre sulla propria uscita il segnale elaborato dal *Baud Rate Generator* necessita che sul suo primo ingresso sia posto un 1 logico, ovvero che la trasmissione sia abilitata attraverso il bit TXEN.

Quando il carattere da trasmettere viene trasferito dal registro parallelo TX-REG al registro seriale TSR, se il bit TXIE è attivo, viene prodotto, mediante la porta AND 1, un segnale di interruzione. Ciò comporterà un salto al vettore di interruzione che, a sua volta, indirizzerà verso la ISR. Sarà possibile valutare la sorgente dell'interruzione testando gli *interrupt flag*. Se l'interruzione sarà stata generata dal blocco di trasmissione asincrona, sarà attivo il bit TXIF, che verrà resettato dal caricamento del prossimo carattere nel registro TXREG.

[15]In realtà il bit di TRMT non diventa attivo quando si svuota il registro TRS, ma quando è stato trasmesso l'ultimo bit del carattere, ovvero il bit di Stop.

Si noti che l'interruzione avviene *all'atto del trasferimento del carattere da trasmettere dal registro TXREG al registro TSR* e non a carattere trasmesso. L'interruzione ha quindi il significato di invito al caricamento di un nuovo carattere.

4.2.9.3 La trasmissione *back to back*

Una caratteristica interessante del blocco di trasmissione è la funzionalità *back to back*. Essa permette la trasmissione ininterrotta di caratteri, senza alcuna pausa fra il bit di Stop di un carattere ed il bit di Start del carattere successivo. I diagrammi temporali esplicativi della funzionalità sono illustrati in fig. 4.50.

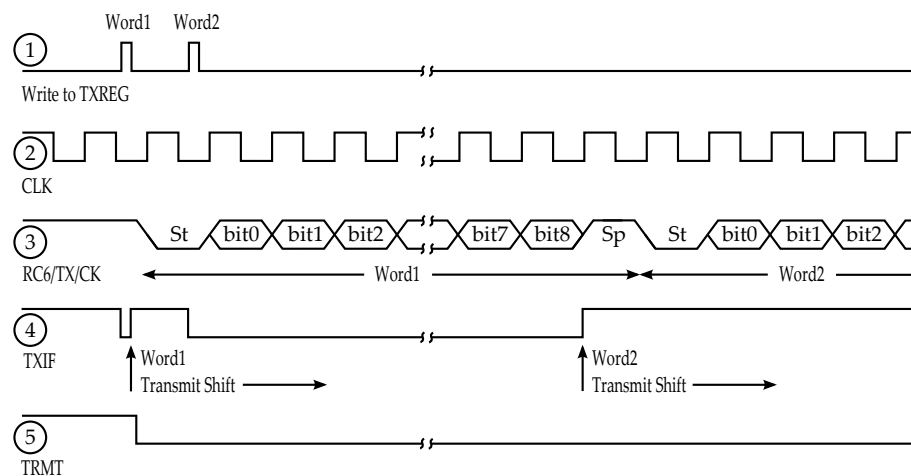



Figura 4.50: Trasmissione *back to back*

Il diagramma 1 evidenzia la scrittura del carattere da trasmettere nel registro parallelo TXREG. Si suppone che prima della scrittura del Word1 la comunicazione sia in stato di *idle*, ovvero in attesa di trasmissione. Contestualmente alla scrittura del Word1 nel registro TXREG si ha anche l'attivazione del *flag* di interruzione (vedi diagramma 4) e conseguente richiamo del vettore di interruzione.

Subito dopo l'attivazione del bit TXIF il segnale TRMT viene posto a 0 logico (vedi diagramma 5), indicando che il *Transmit Shift Register* è stato caricato con il dato da trasmettere serialmente. Contestualmente ha inizio lo shift e viene posto sul pin di uscita il bit di Start (vedi diagramma 3). Lo shift inizia sul primo fronte di salita del segnale CLK, prodotto dal BRG, dopo il caricamento del carattere nel registro TXREG (vedi diagramma 2).

Contestualmente all'attivazione del bit TXIF e al richiamo del vettore di interruzione si entra nella *Interrupt Service Routine*. All'interno della ISR il programma deve riconoscere la periferica che ha lanciato l'interruzione e asservirla. Essendo attivo il bit TXIF è possibile individuare la sorgente dell'interrupt: il carattere caricato nel registro parallelo TXREG è stato trasferito nel registro seriale TSR. La notifica di tale evento permette il caricamento del successivo carattere (Word2).

Quindi il caricamento del secondo carattere avviene con larghissimo anticipo sulla fine della trasmissione del primo carattere. Ciò provoca un secondo azzeramento del bit TXIF (vedi diagramma 4) che verrà automaticamente cancellato (si ricorda che il bit è a sola lettura) solo quando verrà caricato il secondo carattere nel registro TSR, ovvero in fase di shift del bit di Stop del primo carattere.

Il programmatore fa quindi bene a sfruttare questa caratteristica dell'USART, abilitando l'interruzione sul caricamento del carattere nel registro parallelo (TXIE) e caricando immediatamente, nella ISR, il carattere successivo.

Si noti, però, che il richiamo della ISR in seguito al caricamento dell'ultimo carattere nel TXREG non rappresenta la fine della trasmissione. L'ultimo carattere può considerarsi trasmesso solo quando il bit TRMT torna alto. Ciò è particolarmente importante durante una trasmissione RS485, ove è fondamentale gestire oculatamente il tempo di occupazione del bus.

4.2.9.4 La programmazione del trasmettitore

Microchip propone una semplice scaletta, seguendo la quale si pone efficacemente il trasmettitore in grado di espletare le proprie funzioni. I passi da seguire durante la programmazione sono i seguenti:

1. porre nel registro SPBRG l'appropriato valore derivante dal calcolo della 4.4 a pagina 124, che determina il *baud rate* voluto. A tal fine, si deve anche provvedere al valore del bit BRGH. Se si desidera una velocità di comunicazione alta, è bene porre tale bit a 1. Si faccia riferimento alla tabella di fig. 4.40 a pagina 123, verificando che l'errore relativo percentuale sl *baud rate* sia accettabile. Se non si è usata una frequenza di clock standard, l'errore va calcolato manualmente. Dettagli sono forniti nella sezione 4.2.1;
2. abilitare la comunicazione asincrona azzerando il bit SYNC del registro TXSTA e ponendo a 1 il bit SPEN del registro RCSTA;
3. se si desiderano utilizzare le interruzioni in trasmissione si deve porre a 1 il bit TXIE. In tal modo verrà lanciata un'interruzione ogni volta che verrà trasferito un carattere dal registro parallelo TXREG al registro seriale TSR;
4. se si desidera la trasmissione a 9 bit (ad esempio, per trasmetterci la parità) si deve porre a 1 il bit TX9;
5. abilitare la trasmissione settando il bit TXEN. Tale procedura setta automaticamente anche il bit TXIF;
6. se si è scelta la trasmissione a 9 bit, si deve calcolare e caricare in TX9D il nono bit. Se il nono bit contiene informazioni sulla parità il calcolo da fare è dato proprio dal calcolo della parità pari o dispari scelta per la comunicazione. A tal proposito si veda la sezione 4.2.9.1;
7. caricare in TXREG il carattere da trasmettere. Tale azione dà inizio alla trasmissione vera e propria;
8. se si è scelto di trasmettere sfruttando le interruzioni, ci si deve assicurare che i bit PEIE e GIE del registro INTCON siano posti a 1.

In fig. 4.51 sono evidenziati i registri che influenzano la programmazione del blocco di trasmissione dell'USART.

Address	Name	bit 7	bit 6	bit 5	bit 4	bit 3	bit 2	bit 1	bit 0	Value on POR, BOR	Value on all other RESETS
98h	TXSTA	CSRC	TX9	TXEN	SYNC	/	BRGH	TRMT	TX9D	0000 -010	0000 -010
18h	RCSTA	SPEN	RX9	SREN	CREN	ADDEN	FERR	OERR	RX9D	0000 000x	0000 000x
99h	SPBRG	Baud Rate Generator Register								0000 0000	0000 0000
0Bh, 10Bh 8Bh, 18Bh	INTCON	GIE	PEIE	T0IE	INTE	RBIE	T0IF	INTF	RBIF	0000 000x	0000 000u
8Ch	PIE1	PSPIE	ADIE	RCIE	TXIE	SSPIE	CCP1IE	TMR2IE	TMR1IE	0000 0000	0000 0000
0Ch	PIR1	PSPIF	ADIF	RCIF	TXIF	SSPIF	CCP1IF	TMR2IF	TMR1IF	0000 0000	0000 0000
19h	TXREG	USART Transmit Register								0000 0000	0000 0000

Legend: x = unknown, u = unchanged, / = unimplemented locations read as 0
Shaded cells are not used for asynchronous transmission

Figura 4.51: Registri coinvolti nella programmazione della trasmissione

4.2.9.5 Il blocco di ricezione

Il blocco di ricezione asincrona è l'ultimo dei blocchi dell'USART. Lo schema a blocchi è illustrato in fig. 4.52.

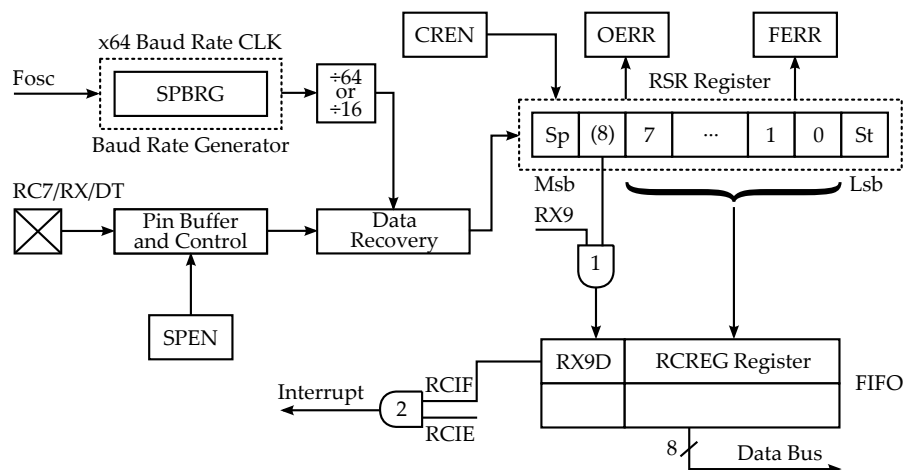



Figura 4.52: Blocco di ricezione asincrona

La ricezione asincrona viene abilitata attraverso il bit CREN del registro RCSTA. Tale bit abilita a sua volta il registro seriale di ricezione RSR (si no-

ti che il registro non è mappato in memoria) che si pone in attesa dei bit che si presentano serialmente al pin di ingresso RC7/RX/DT. Il registro seriale sa se la ricezione avviene su 8 oppure su 9 bit[16] ed è quindi in grado di riconoscere il bit di Stop. Quando tale bit viene ricevuto, il carattere viene trasferito dal registro seriale RSR al registro paralello RCREG, da dove può essere letto attraverso il bus dei dati.

Nel momento in cui termina il trasferimento del carattere dal registro sertiale a quello parallelo viene settato il bit RXIF del registro PIR1 e viene contestualmente lanciata opportuna interruzione se i bit RCIE, PEIE e GIE erano stato correttamente abilitati.

Nella ISR il carattere appena ricevuto deve essere letto, eseguendo una lettura del registro RCREG. Questa azione provoca il reset del bit RCIF.

Si noti che, in realtà, il registro RCREG è un registro FIFO a due livelli. Ciò significa che possono essere ivi memorizzati fino a due caratteri, mentre un terzo è in arrivo nel registro seriale RSR. Nel caso in cui il terzo carattere sia pronto prima che il FIFO venga almeno parzialmente svuotato si produce un errore di *overflow* con settaggio del bit di OERR del registro RCSTA. In questo caso il carattere appena ricevuto dal registro seriale viene perso e vengono mantenuti i caratteri memorizzati nel FIFO.

Da questo momento in poi non vengono accettati altri caratteri dal registro seriale finché non viene resettato il bit di OERR. Essendo, però, tale bit a sola lettura, ciò deve avvenire mediante reset dell'intero blocco di ricezione, eseguito, naturalmente, dopo aver svuotato il FIFO. Si deve quindi procedere nel reset del bit di CREN del registro RCSTA e nel successivo set. Tale azione disabilita/riabilita il ricevitore e azzera eventuali errori di *overflow*.

Il bit di FERR del registro RCSTA è settato quando il bit di Stop è trovato a livello logico 0. La lettura del registro RCREG, oltre a permettere la lettura del carattere arrivato, permette anche il caricamento dei relativi valori di RX9D e FERR. E' quindi di fondamentale importanza che, all'arrivo di un nuovo carattere, il registro RCSTA venga letto *prima* del registro RCREG, dato che la lettura di RCREG sovrascrive i vecchi valori di FERR e RX9D. In tal modo è possibile sapere se il carattere è affetto da errore di *framing* oppure di parità. Verificata la bontà o meno del carattere, il registro RCREG *va comunque* svuotato, indipendentemente se il carattere presente nel registro parallelo è corretto o meno.

Si solleva un'altra nota importante, relativa ad un blocco della fig. 4.52 nella pagina precedente. Detta figura è presente nel *datasheet* ufficiale del PIC16F87x. All'uscita del *Baud Rate Generator* è posto un blocco di divisione x64 oppure x16. Si tratta probabilmente di una svista, dato che il divisore dovrebbe operare la divisione x64, in modo da fornire il segnale CLK e la divisione x4 (non x16!) in modo da fornire il segnale necessario alla sincronizzazione del bit di Start.

La confusione nasce probabilmente a causa di una ristretta famiglia di processori (PIC16C63, PIC16C65, PIC16C65A, PIC16C73, PIC16C73A, PIC16C74,

[16]Dallo schema a blocchi non appare in maniera evidente se la ricezione avviene a 8 oppure a 9 bit. In realtà il bit RX9 è letto anche dal circuito di *sampling* del dato seriale, che è quindi in grado di distinguere un bit di parità a 1 dal bit di Stop.

PIC16C74A) che utilizzano effettivamente per la sincronizzazione un clock 4 volte più veloce del segnale CLK (quindi diviso x16 rispetto a quello fornito dal BRG). Si ritiene, quindi, che il blocco divida **x64 *e* x4** e non **x64 *o* x16**.

In fig. 4.53 sono evidenziati i grafici correlati ad una ricezione di tre caratteri senza tempestivo svuotamento del RCREG e conseguente *overflow*.

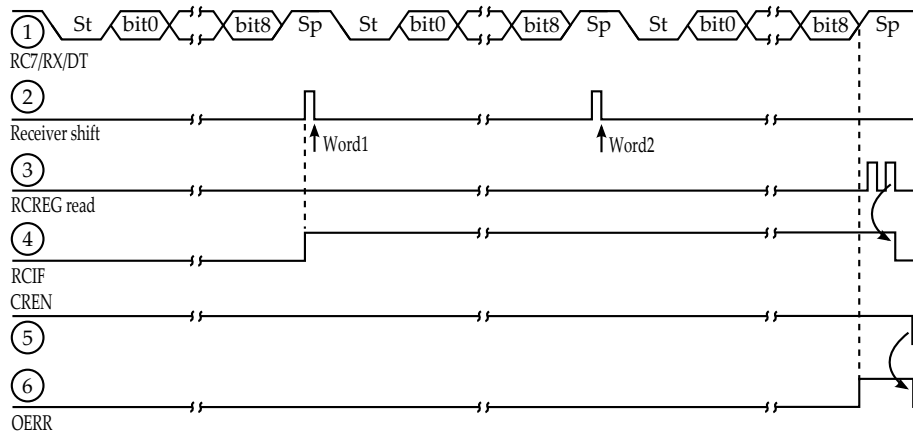



Figura 4.53: Ricezione asincrona con *overflow*

Al completamento della ricezione del primo e del secondo carattere lo shift register trasferisce nel RCREG (che, si ricorda è un FIFO a due livelli) i nove (eventuali) bit ricevuti (vedi diagramma 2). Sul primo trasferimento viene generato un RCIF (vedi diagramma 4), che tornerà disattivo solamente quando il FIFO verrà completamente svuotato (con due letture: vedi diagramma 3).

Alla fine della ricezione del terzo carattere viene generato un errore di *overrun* (vedi diagramma 6), che può essere azzerato solamente disabilitando il blocco di ricezione dell'USART (vedi diagramma 5).

4.2.9.6 La programmazione del ricevitore

Anche per la programmazione del blocco di ricezione Microchip propone una semplice scalettai. I passi da seguire durante la programmazione sono i seguenti:

1. porre nel registro SPBRG l'appropriato valore derivante dal calcolo della 4.4 a pagina 124, che determina il *baud rate* voluto. A tal fine, si deve anche provvedere al valore del bit BRGH. Se si desidera una velocità di comunicazione alta, è bene porre tale bit a 1. Si faccia riferimento alla tabella di fig. 4.40 a pagina 123, verificando che l'errore relativo percentuale sl *baud rate* sia accettabile. Se non si è usata una frequenza di clock standard, l'errore va calcolato manualmente. Dettagli sono forniti nella sezione 4.2.1;
2. abilitare la comunicazione asincrona azzerando il bit SYNC del registro TXSTA e ponendo a 1 il bit SPEN del registro RCSTA;

3. se si desiderano utilizzare le interruzioni in ricezione si deve porre a 1 il bit RCIE. In tal modo verrà lanciata un'interruzione ogni volta che verrà trasferito un carattere dal registro seriale RSR al registro parallelo RCREG;
4. se si desidera la ricezione a 9 bit (ad esempio, per ricevere la parità) si deve porre a 1 il bit RX9;
5. abilitare la ricezione settando il bit CREN;
6. il bit RCIF del registro PIR1 viene settato quando si completa una ricezione con il riconoscimento del bit di Stop. Se è abilitato il bit RCIE del registro PIE1, viene contestualmente lanciata una interruzione;
7. leggere il registro RCSTA per acquisire il nono bit e valutare la presenza di errori di *overrun*, *framing* ed eventualmente di parità;
8. leggere il registro RCREG, indipendentemente dal fatto che nella lettura del registro RCSTA si siano rilevati errori o meno. RCREG va comunque svuotato;
9. se sono stati rilevati errori in ricezione si deve disabilitare la ricezione azzerando momentaneamente il bit CREN del registro RCSTA;
10. se si è scelto di trasmettere sfruttando le interruzioni, ci si deve assicurare che i bit PEIE e GIE del registro INTCON siano posti a 1.

In fig. 4.54 sono evidenziati i registri che influenzano la programmazione del blocco di ricezione dell'USART.

Address	Name	bit 7	bit 6	bit 5	bit 4	bit 3	bit 2	bit 1	bit 0	Value on POR, BOR	Value on all other RESETS
98h	TXSTA	CSRC	TX9	TXEN	SYNC	/	BRGH	TRMT	TX9D	0000 -010	0000 -010
18h	RCSTA	SPEN	RX9	SREN	CREN	ADDEN	FERR	OERR	RX9D	0000 000x	0000 000x
99h	SPBRG	Baud Rate Generator Register								0000 0000	0000 0000
0Bh, 10Bh 8Bh, 18Bh	INTCON	GIE	PEIE	T0IE	INTE	RBIE	T0IF	INTF	RBIF	0000 000x	0000 000u
8Ch	PIE1	PSPIE	ADIE	RCIE	TXIE	SSPIE	CCP1IE	TMR2IE	TMR1IE	0000 0000	0000 0000
0Ch	PIR1	PSPIF	ADIF	RCIF	TXIF	SSPIF	CCP1IF	TMR2IF	TMR1IF	0000 0000	0000 0000
1Ah	RCREG	USART Receive Register								0000 0000	0000 0000

Legend: x = unknown, u = unchanged, / = unimplemented locations read as 0
Shaded cells are not used for asynchronous reception

Figura 4.54: Registri coinvolti nella programmazione della ricezione

Parte II

L'ASSEMBLY DEL PIC

Capitolo 5

L'assembly del PIC16F877

Il presente capitolo è interamente dedicato al linguaggio *assembly* del PIC16F877A. Detto linguaggio è identico per tutti i microcontrollori della serie 16F e presenta molte somiglianze con i linguaggi assembly delle altre famiglie dei PIC.

Lo studente pratico di linguaggi evoluti (C, Pascal, OOP, Java, ecc.) tarrà sicuramente giovamento da tali conoscenze ma non dovrà aspettarsi la stessa potenza di calcolo e flessibilità.

In generale, l'assembly è un linguaggio *line oriented*, il che significa che la singola istruzione viene interrotta quando si va "a capo" in una linea nuova. Nei linguaggi ad alto livello ciò non succede e le singole istruzioni possono essere scritte su più linee senza essere interrotte.

Un'altra differenza piuttosto significativa consiste nel fatto che le istruzioni sono *mnemoniche*, sono cioè scritte in modo tale da ricorda la loro funzione. Nei linguaggi ad alto livello, solitamente, le istruzioni sono esplicitate in maniera chiara ed estesa (si pensi alle istruzioni `if` oppure `while`).

Una terza differenza che di solito colpisce molto gli studenti è l'assenza del calcolo in virgola mobile o delle semplici operazioni aritmetiche.

Lo studente deve quindi rendersi conto che l'assembly è un linguaggio assolutamente elementare e molto vicino al linguaggio macchina del microcontrollore. Semplici calcoli come 3.14/ 0.5 sono di una complessità notevole e di pertinenza dei programmatori più esperti.

Per contro, però, l'assembly offre una notevolissima compattezza ed un accesso alle singole risorse del controllore assolutamente chirurgica e non può rimanere sconosciuto a chi volesse avvicinarsi al mondo dei microprocessori.

Le singole istruzioni del PIC verranno esaminate e commentate una ad una in ordine alfabetico. Prima però è necessario fornire alcune nozioni di tipo generale.

5.1 Le istruzioni del PIC16F877

Le istruzioni assembly del PIC16F877A sono raggruppate sostanzialmente in tre categorie:

- istruzioni orientate al byte;
- istruzioni orientate al bit;
- istruzioni di controllo e contenenti costanti.

Nelle istruzioni orientate al byte il simbolo `f` rappresenta il *file register*, ovvero un qualsiasi registro contenuto in uno dei banchi di memoria dati del PIC. Il simbolo `d` rappresenta, invece, la destinazione del risultato dell'operazione eseguita dall'istruzione: se `d` vale 0 il risultato è posto nel registro W, mentre se `d` vale 1 il risultato è posto nello stesso registro indicato da `f`.

Quindi

Listing 5.1: Esempio di istruzione *byte oriented*

```
movf    f,d     ;Sintassi istruzione
```

sta ad indicare la sintassi dell'istruzione `movf`, dove `f` rappresenta un generico registro e `d` la destinazione. Esempi validi sono i seguenti:

Listing 5.2: Due esempi pratici

```
movf    pippo,0 ;Sposta il contenuto di pippo in W
movf    pippo,W ;Stessa sintassi, ma piu' leggibile
```

Nelle istruzioni orientate al bit si incontra il simbolo `b` che rappresenta un valore costante compreso fra 0 e 7 ed indica sempre il bit `b` di un byte.

Quindi

Listing 5.3: Esempio di istruzione *bit oriented*

```
bcf f,b         ;Sintassi istruzione di bit clear
```

sta ad indicare la sintassi dell'istruzione `bcf`, dove `f` rappresenta un generico registro e `b` una costante compresa fra 0 e 7 compresi, indicante un bit del registro `f`. Esempi validi sono i seguenti:

Listing 5.4: Un esempio pratico

```
bcf pippo,4     ;Resetta il bit 4 di pippo
```

Nelle istruzioni di controllo e contenenti costanti si ritrova il simbolo `k`. Esso rappresenta una costante di 8 oppure 11 bit a seconda dell'istruzione coinvolta.

Quindi

Listing 5.5: Esempio di istruzione con costante

```
movlw   k       ;Sintassi istruzione di caricamento
goto    k       ;Sintassi istruzione di salto
```

stanno ad indicare la sintassi delle istruzioni `movlw` e `goto`, dove `k` rappresenta una generica costante rispettivamente di 8 e 11 bit. Esempi validi sono i seguenti:

Listing 5.6: Altri due esempi pratici

```
movlw   kPippo  ;Carica il valore kPippo in W
goto    kPluto  ;Salta all'istruzione indicata da kPluto
```

5.1.1 ADDLW

ADDLW	Add Literal and W
Syntax:	[*label*] ADDLW k
Operands:	$0 \leq k \leq 255$
Operation:	(W) + k $\rightarrow$ (W)
Status Affected:	C, DC, Z
Description:	The contents of the W register are added to the eight bit literal 'k' and the result is placed in the W register.

Commento ed Esempi

Si supponga che prima dell'istruzione si abbia `W = 7`.

Listing 5.7: Esempio 1 di ADDLW

```
addlw   9       ;Somma al contenuto di W il valore 9
```

Al termine dell'istruzione il risultato, ossia 16, è stato posto in `W`. Si noti che essendoci stato un riporto fra nibble basso e nibble alto di `W` nel registro di stato viene settato il bit di `DC`, mentre nè il bit di zero nè quello di riporto vengono settati non essondosi verificato nessuno dei due casi.

Più interessante è il seguente esempio. Si supponga che `W = 129`.

Listing 5.8: Esempio 2 di ADDLW

```
addlw   5       ;Dopo la somma W = 134 e C = 0
```

Al termine dell'operazione il registro di lavoro contiene 134. Potrebbe sembrare ovvio, ma non lo è. Una siffatta operazione è, infatti, valida sia che gli addendi siano *signed* che *unsigned*. Se si considera in complemento a due il primo addendo, esso assume il valore -127_{10} e l'operazione aritmetica $-127_{10} + 5_{10}$ dà -122_{10}, ovvero $0x86$ in rappresentazione in complemento a due e 134_{10} in base 10.

Un altro esempio da studiare è il seguente. Si supponga che `W = 83`.

Listing 5.9: Esempio 3 di ADDLW

```
addlw   48      ;Dopo la somma W = 131 (0x83) e C = 1
```

Se si considerano gli addendi come *unsigned* il risultato è corretto dato che $83 + 48 = 131$. Se, però, si interpreta l'operazione aritmetica in complemento a due, il risultato è errato: $83 + 48 = -125$. Ciò è evidente se si considera che il risultato ha segno opposto agli addendi.

Anche il prossimo esempio non è rassicurante. Si supponga che `W = 173`

Listing 5.10: Esempio 4 di ADDLW

```
addlw   214     ;Dopo la somma W = 131 (0x83) e C = 1
```

Interpretando l'operazione aritmetica non in complemento a due il risultato è errato: $173 + 214 = 131$ (infatti il Carry testimonia un supero di capacità). Bisognerebbe sommare il peso del nono bit ($2^8 = 256$) per ottenere un risultato corretto.

In complemento a due, invece, il risultato è assolutamente corretto, dato che $(-83_{10}) + (-42_{10}) = -125_{10}$, ovvero 0xAD+0xD6=0x83. La correttezza del risultato è avvalorata dalla concordanza dei segni fra operandi e risultato, come pure dal fatto che, nella somma binaria, si ha riporto sia sulla colonna del segno che su quella del Carry (vedi il paragrafo **??** a pagina ?? ed in particolare la relazione **??** a pagina ??).

Appare quindi evidente l'importanza di interpretare correttamente i risultati forniti dall'ALU, con particolare riferimento al tipo di operazione aritmetica effettuata, se in complemento a due o meno.

5.1.2 ADDWF

ADDWF	Add W and f
Syntax:	[*label*] ADDWF f,d
Operands:	$0 \leq f \leq 127$ $d \in [0,1]$
Operation:	(W) + (f) $\rightarrow$ (destination)
Status Affected:	C, DC, Z
Description:	The contents of the W register are added to the 'f' register. If 'd' is 0, the result is stored in the W register. If 'd' is 1, the result is stored back in the 'f' register.

Commento ed Esempi

Si supponga che prima dell'istruzione si abbia `W = 7` e che `Pippo = 9`.

Listing 5.11: Esempio 1 di ADDWF

```
addwf    Pippo  ;Somma a W il contenuto di Pippo
                ;Il risultato e' posto in Pippo
```

Si noti che il risultato viene posto in `Pippo`, per cui `Pippo = 16 (0x10)`.

Si supponga sempre che `W = 7` e che `Pippo = 9`.

Listing 5.12: Esempio 2 di ADDWF

```
addwf    Pippo,W;Somma a W il contenuto di Pippo
                ;Il risultato e' posto in W
```

In questo caso il risultato viene posto in `W`, per cui `W = 16 (0x10)`.

L'intera analisi fatta per l'istruzione `addlw` è valida anche per la presente istruzione.

5.1.3 ANDLW

ANDLW	AND Literal with W
Syntax:	[*label*] ANDLW k
Operands:	$0 \leq k \leq 255$
Operation:	(W) .AND. k $\rightarrow$ (W)
Status Affected:	Z
Description:	The contents of the W register are AND'ed with the eight bit literal 'k' and the result is placed in the W register.

Commento ed Esempi

Si supponga che prima dell'istruzione si abbia `W = 0xFF`.

Listing 5.13: Esempio 1 di ANDLW

```
andlw    0x55   ;Esegue l'AND logico fra W e 0x55
```

Siccome, dopo l'esecuzione dell'istruzione, si ha `W = 0x55`, il *flag* di Zero vale `Z = 0`.

Se, invece, si suppone `W = 0x55`, l'istruzione

Listing 5.14: Esempio 2 di ANDLW

```
andlw    0xAA   ;Esegue l'AND logico fra W e 0xAA
```

dà come risultato `W = 0`, per cui `Z = 1`.

Si supponga di non conoscere il contenuto di `W` e di voler sapere se i bit 2 e 5 di tale registro sono entrambi a 0:

Listing 5.15: Esempio 3 di ANDLW

```
andlw    B'001001000';Maschera con i bit 2 e 5
```

L'operando dell'istruzione dell'esempio 3 è detto **maschera** e permette di valutare, insieme all'istruzione `andlw` se i bit posti a 1 della maschera sono entrambi a 0 o meno. Se sono entrambi a 0 si ha `Z = 1`, altrimenti `Z = 0`.

5.1.4 ANDWF

ANDWF	AND W with f
Syntax:	[*label*] ANDWF f,d
Operands:	$0 \leq f \leq 127$ $d \in [0,1]$
Operation:	(W) .AND. (f) $\rightarrow$ (destination)
Status Affected:	Z
Description:	AND the W register with register 'f'. If 'd' is 0, the result is stored in the W register. If 'd' is 1, the result is stored back in the 'f' register.

Commento ed Esempi

Si supponga che `W = 0xFF` e che `Pippo = 0x55`.

Listing 5.16: Esempio 1 di ANDWF

```
andwf    Pippo  ;Esegue l'AND logico fra W e Pippo.
                ;Il risultato e' posto in Pippo.
```

Si noti che il risultato viene posto in `Pippo`, per cui `Pippo` non cambia e `Z = 0`.

Si supponga `W = 0xFF` e `Pippo = 0x55`.

Listing 5.17: Esempio 2 di ANDWF

```
andwf    Pippo,W;Esegue l'AND logico fra il contenuto
                ;W e Pippo. Il risultato e' posto in W.
```

In questo caso il risultato viene posto in `W`, per cui si ha che `W = 0x55` e `Pippo = 0x55`. Il *flag* di Zero non viene settato.

L'analisi fatta per l'istruzione `andlw` è valida anche per la presente istruzione.

5.1.5 BCF

BCF	Bit Clear f
Syntax:	[*label*] BCF f,b
Operands:	$0 \leq f \leq 127$
	$0 \leq b \leq 7$
Operation:	$0 \rightarrow$ (f<b>)
Status Affected:	None
Description:	Bit 'b' in register 'f' is cleared.

Commento ed Esempi

Si supponga `Pippo = 0x55`.

Listing 5.18: Esempio 1 di BCF

```
bcf     Pippo,0;Azzera il bit 0 di Pippo.
               ;Dopo l'operazione, Pippo = 0x54.
```

Il secondo operando può essere solo una costante nel *range* 0-7.

Si noti che il primo operando non può essere il registro `W`, ma solo un *file register*. Si supponga, ad esempio, che `W = 0x55`. L'istruzione

Listing 5.19: Esempio 2 di BCF

```
bcf     W,0    ;Non azzera il bit 0 di W.
```

lascia inalterato il contenuto di `W`.

5.1.6 BSF

BSF	Bit Set f
Syntax:	[*label*] BSF f,b
Operands:	$0 \leq f \leq 127$
	$0 \leq b \leq 7$
Operation:	$1 \rightarrow (f<b>)$
Status Affected:	None
Description:	Bit 'b' in register 'f' is set.

Commento ed Esempi

E' l'istruzione duale della precedente. Si supponga, ad esempio, `Pippo = 0x55`.

Listing 5.20: Esempio 1 di BSF

```
    bcf     Pippo,1;Pone a 1 il bit 1 di Pippo.
                   ;Dopo l'operazione, Pippo = 0x57.
```

Il secondo operando può essere solo una costante nel *range* 0-7. Anche in questo caso non può essere usato il registro `W` come primo operando.

5.1.7 BTFSS

BTFSS	Bit Test f, Skip if Set
Syntax:	[*label*] BTFSS f,b
Operands:	$0 \leq f \leq 127$ $0 \leq b \leq 7$
Operation:	skip if (f<b>) = 1
Status Affected:	None
Description:	If bit 'b' in register 'f' is 0, the next instruction is executed. If bit 'b' is 1, the the next instruction is discarded and a NOP is executed instead, making this a $2T_{CY}$ instruction.

Commento ed Esempi

E' l'istruzione che permette di implementare la selezione. La condizione booleana è rappresentata dallo stato di un bit di un *file register*. Si supponga `Pippo = 0x55`.

Listing 5.21: Esempio 1 di BTFSS

```
btfss   Pippo,0;Se Pippo<0>=1, esegue lo skip
bsf     Pippo,7;Questa istruzione non viene eseguita
```

La `btfss` testa il bit indicato dal secondo operando (che deve essere una costante nel *range* 0-7) del *file register* indicato nel primo operando: se vale 1 l'istruzione che segue la `btfss` viene saltata (*skipped*). Si noti che, in tal caso, la `btfss` modifica il PC, per cui l'istruzione dura due cicli macchina (vedi il paragrafo 1.5 a pagina 11).

Le istruzioni indicate dall'esempio 5.21 potrebbero essere tradotte in linguaggio C nel seguente modo:

Listing 5.22: Esempio 2 di BTFSS in C

```
if (!(Pippo & 0x01))
   Pippo |= 0x80;
```

Anche in questo caso, se `Pippo = 0x55` prima dell'`if`, il corpo dell'`if` non viene eseguito.

5.1.8 BTFSC

BTFSC	Bit Test f, Skip if Clear
Syntax:	[*label*] BTFSC f,b
Operands:	$0 \leq f \leq 127$ $0 \leq b \leq 7$
Operation:	skip if (f<b>) = 0
Status Affected:	None
Description:	If bit ‘b’ in register ‘f’ is 1, the next instruction is executed. If bit ‘b’ is 0, the the next instruction is discarded and a NOP is executed instead, making this a $2T_{CY}$ instruction.

Commento ed Esempi

E’ l’istruzione duale della precedente. La condizione booleana è ancora rappresentata dallo stato di un bit di un *file register*, ma stavolta lo *skip* avviene se il bit è posto a 0. Si supponga `Pippo = 0xAA`.

Listing 5.23: Esempio 1 di BTFSC

```
btfsc   Pippo,0;Se Pippo<0>=0, esegue lo skip
bsf     Pippo,7;Questa istruzione non viene eseguita
```

La `btfsc` testa il bit indicato dal secondo operando (che deve essere una costante nel *range* 0-7) del *file register* indicato nel primo operando: se vale 0 l’istruzione che segue la `btfsc` viene saltata (*skipped*). Si noti che, in tal caso, la `btfsc` modifica il PC, per cui l’istruzione dura due cicli macchina.

Le istruzioni indicate dall’esempio 5.23 potrebbero essere tradotte in linguaggio C nel seguente modo:

Listing 5.24: Esempio 2 di BTFSC in C

```
if (Pippo & 0x01)
   Pippo |= 0x80;
```

Anche in questo caso, se `Pippo = 0xAA` prima dell’`if`, il corpo dell’`if` non viene eseguito.

5.1.9 CALL

CALL	Call Subroutine
Syntax:	[*label*] CALL k
Operands:	$0 \leq k \leq 2047$
Operation:	(PC) + 1 → TOS, k → PC<10:0>, (PCLATH<4:3>) → PC<12:11>
Status Affected:	None
Description:	Call Subroutine. First, return address (PC+1) is pushed onto the Top Of the Stack. The eleven-bit immediate address is loaded into PC bits <10:0>. The upper bits of the PC are loaded from PCLATH. CALL is a two-cycle instruction.

Commento ed Esempi

Quest'istruzione permette di richiamare una subroutine. Si veda il seguente codice:

Listing 5.25: Esempio 1 di CALL

```
        movlw   0x55    ;W = 0x55
        call    IncW    ;Dopo la subroutine W = 0x56
        addlw   2       ;W = 0x58
        call    IncW    ;Dopo la subroutine W = 0x59
        addlw   3       ;W = 0x5C
        ...             ;Seguono altre istruzioni
        ...
IncW:   ;Subroutine IncW. Essa ha il compito di incrementare
        ;il registro W
        addlw   1       ;Incrementa il valore di W
        return          ;Ritorna all'istruzione seguente la
                        ;chiamata alla subroutine
```

La prima volta che viene richiamata la subroutine `IncW` il registro `W` contiene il valore 0x55. La `IncW` incrementa tale valore e torna all'istruzione seguente la `call` (ossia la `addlw 2`).

La seconda volta che viene richiamata la subroutine `IncW` il registro `W` contiene il valore 0x58. La `IncW` incrementa nuovamente tale valore e torna all'istruzione seguente la `call` (in questo caso, la `addlw 3`).

Si noti che l'istruzione `call` salva solamente l'indirizzo di ritorno e null'altro. E' quindi compito del programmatore salvare eventualmente lo stato dei registri utilizzati. Tipicamente vengono salvati gli stati dei registri W e STATUS. L'istruzione `call` necessita sempre di due cicli macchina per l'esecuzione.

Per dettagli sulla memorizzazione dell'indirizzo di ritorno si veda il paragrafo 1.7.1 a pagina 18.

La `call` modifica il PC, per cui l'istruzione dura sempre due cicli macchina (vedi il paragrafo 1.5 a pagina 11).

5.1.10 CLRF

CLRF	Clear f
Syntax:	[*label*] CLRF f
Operands:	$0 \leq f \leq 127$
Operation:	$00h \rightarrow (f)$ $1 \rightarrow Z$
Status Affected:	Z
Description:	The contents of register 'f' are cleared and the Z bit is set.

Commento ed Esempi

Si supponga che prima dell'istruzione si abbia `Pippo = 0x55`.

Listing 5.26: Esempio 1 di CLRF

```
clrf    Pippo   ;Dopo l'istruzione Pippo = 0 e Z = 1
```

L'istruzione permette di azzerare il contenuto di un qualsiasi *file register*, settando contemporaneamente il *flag* di Zero. Si noti che l'operando non può essere il registro `W`, ma un *file register*. Se vi è la necessità di azzerare il registro `W`, si vedano il paragrafo seguente ed il paragrafo 5.1.22 a pagina 172.

5.1.11 CLRW

CLRW	Clear W
Syntax:	[*label*] CLRW
Operands:	None
Operation:	00h $\rightarrow$ (W), 1 $\rightarrow$ Z
Status Affected:	Z
Description:	W register is cleared. Z bit is set.

Commento ed Esempi

Si supponga che prima dell'istruzione si abbia `W = 0x55`.

Listing 5.27: Esempio 1 di CLRW

```
clrw            ;Dopo l'istruzione W = 0 e Z = 1
```

L'istruzione permette di azzerare il contenuto del registro `W`, settando contemporaneamente il *flag* di Zero. Se si vuole azzerare il registro `W` ed evitare di modificare il registro di stato si può ricorrere al seguente codice:

Listing 5.28: Esempio 2 di CLRW

```
movlw   0       ;Dopo l'istruzione W = 0 mentre Z non
                ;viene modificato
```

Si veda a tal proposito il paragrafo 5.1.22 a pagina 172.

5.1.12 CLRWDT

CLRWDT	Clear Watchdog Timer
Syntax:	[*label*] CLRWDT
Operands:	None
Operation:	00h $\rightarrow$ WDT, 0 $\rightarrow$ WDT prescaler, 1 $\rightarrow$ $\overline{TO}$, 1 $\rightarrow$ $\overline{PD}$
Status Affected:	$\overline{TO}$, $\overline{PD}$
Description:	CLRWDT instruction resets the Watchdog Timer. It also resets the prescaler of the WDT. Status bits $\overline{TO}$ and $\overline{PD}$ are set.

Commento ed Esempi

L'istruzione permette l'azzeramento del *Watchdog Timer* e del relativo *prescaler*, se abbinato al WDT.

Listing 5.29: Esempio 1 di CLRW

```
clrwdt ;Il WDT ricomincia il conteggio da 0
```

Si noti che, oltre ad azzerare il WDT, l'istruzione `clrwdt` setta anche i bit $\overline{TO}$ e $\overline{PD}$ nel registro di stato. Ciò è necessario perché se il WDT dovesse arrivare a fine conteggio, esso lancerebbe un reset al micro che azzererebbe il PC e ricomincerebbe da capo nell'esecuzione del programma (vedi il paragrafo 1.7.2 a pagina 22). Il micro, però, non sa per quale motivo è stato richiamato il vettore di reset ed è compito del programmatore stabilirlo. La prima cosa da fare è verificare lo stato dei due citati bit, il cui stato è riassunto dalla sottostante tabella:

$\overline{TO}$	$\overline{PD}$	Descrizione
1	1	Dopo un *power up* oppure dopo l'esecuzione dell'istruzione `crlwdt`
1	0	In seguito all'esecuzione dell'istruzione di `sleep` (vedi par. 5.1.30 a pagina 180)
0	1	Intervento del WDT
0	0	Concomitanza dell'intervento del WDT e dello *sleep*

Tabella 5.1: Interpretazione dei bit $\overline{TO}$ e $\overline{PD}$

La prima condizione ($\overline{TO} = 1$ e $\overline{PD} = 1$) è quella di normalità. La si incontra dopo la prima accensione del micro oppure dopo l'esecuzione dell'istruzione `clrwdt`.

La seconda condizione ($\overline{TO} = 1$ e $\overline{PD} = 0$) è quella che si avvera dopo l'esecuzione dell'istruzione `sleep` (vedi). E' importante tenerne traccia per riconoscere la condizione di uscita dallo *sleep*.

La terza condizione ($\overline{TO} = 0$ e $\overline{PD} = 1$) è quella potenzialmente più pericolosa. Essa testimonia l'intervento del WDT e non una nuova accensione. Tale situazione deve essere assolutamente rilevata dal programmatore al fine di porre in sicurezza il sistema (vedi par. 1.10.3 a pagina 33).

L'ultima condizione corrisponde allo scadere del WDT durante uno *stand by*. Ciò indica che il WDT ha "risvegliato" il micro, che costituisce una condizione normale di funzionamento.

E' preciso compito del programmatore testare i suddetti bit nelle prime righe di codice, al fine di valutare se vi è stato un possibile intervento del WDT.

Per ulteriori dettagli sul *watchdog timer* si veda la sezione 1.10.3 a pagina 33. Uteriori dettagli sui bit di $\overline{TO}$ e $\overline{PD}$ sono forniti alla sezione 3.1 a pagina 59.

5.1.13 COMF

COMF	Complement f
Syntax:	[*label*] COMF f
Operands:	$0 \leq f \leq 127$ $d \in [0,1]$
Operation:	$(\bar{f}) \rightarrow$ (destination)
Status Affected:	Z
Description:	The contents of register 'f' are complemented. If 'd' is 0, the result is stored in the W register. If 'd' is 1, the result is stored back in the 'f' register.

Commento ed Esempi

L'istruzione esegue il complemento a 1 del *file register* indicato come argomento. Si supponga che, prima dell'esecuzione dell'istruzione indicata di seguito, si abbia `Pippo = B'10101010'`.

Listing 5.30: Esempio 1 di COMF

```
        comf    Pippo ;Dopo l'istruzione Pippo = B'01010101'
```

Si tratta evidentemente della negazione del contenuto del *file register*. Infatti altri assembler indicano tale istruzione con il mnemonico `neg`.

Naturalmente è possibile porre il risultato della negazione in `W`, lasciando inalterato il contenuto del *file register*, come mostrato nel seguente esempio (supponendo sempre `Pippo = B'10101010'` prima dell'esecuzione dell'istruzione).

Listing 5.31: Esempio 2 di COMF

```
        comf    Pippo,W ;Dopo l'istruzione W = B'01010101' e
                        ;Pippo = B'10101010'
```

Si noti che se il risultato dovesse valere zero, viene settato il *flag* di Zero.

5.1.14 DECF

DECF	Decrement f
Syntax:	[*label*] DECF f,d
Operands:	$0 \leq f \leq 127$ $d \in [0,1]$
Operation:	(f) - 1 $\rightarrow$ (destination)
Status Affected:	Z
Description:	The contents of register ‘f’ are decremented. If ‘d’ is 0, the result is stored in the W register. If ‘d’ is 1, the result is stored back in the ‘f’ register.

Commento ed Esempi

L’istruzione esegue il decremento del *file register* indicato in argomento. Anche in questo caso è istruttivo analizzare cosa succede se si decrementa un numero e lo si interpreta in complemento a due piuttosto che senza segno. Si veda il seguente esempio, supponendo inizialmente `Pippo = B’00000010’`.

Listing 5.32: Esempio 1 di DECF

```
1.      decf    Pippo ;Dopo l’istruzione Pippo = B’00000001’
2.      decf    Pippo ;Dopo l’istruzione Pippo = B’00000000’
3.      decf    Pippo ;Dopo l’istruzione Pippo = B’11111111’
4.      decf    Pippo ;Dopo l’istruzione Pippo = B’11111110’
```

Prima dell’esecuzione della prima istruzione si aveva `Pippo = 0x02`, ossia il valore positivo 2. Tale valore viene interpretato allo stesso modo sia in complemento a due sia in notazione *unsigned*, dato che il valore è positivo.

Dopo l’esecuzione dell’istruzione 1. si ha, infatti, `Pippo = 0x01`, ossia 1. Il valore del registro è stato decrementato di una unità.

Dopo l’esecuzione dell’istruzione 2. si ha `Pippo = 0x00`, con settaggio del *flag* di Zero, dato che il risultato del decremento ha prodotto il valore 0.

Piuttosto interessante è ciò che accade durante l’esecuzione dell’istruzione 3. Il valore che il registro assume, dopo il decremento, è `Pippo = 0xFF`. Tale valore è interpretabile come 255 se è considerato senza segno, oppure -1 se viene interpretato in complemento a due. In entrambi i casi il risultato è assolutamente logico e rispettoso dell’ortodossia aritmetica.

Decrementando ulteriormente detto valore (istruzione 4.) si ottiene 254 se si interpreta il valore come rappresentato in formato *unsigned*, oppure si ottiene -2 se si interpreta il risultato in complemento a due.

L’istruzione `decf` funziona quindi perfettamente si in notazione senza segno che in complemento a due.

5.1.15 DECFSZ

DECFSZ	Decrement f, Skip if 0
Syntax:	[*label*] DECFSZ f,d
Operands:	$0 \leq f \leq 127$ $d \in [0,1]$
Operation:	(f) - 1 $\rightarrow$ (destination), skip if result = 0
Status Affected:	None
Description:	The contents of register 'f' are decremented. If 'd' is 0, the result is stored in the W register. If 'd' is 1, the result is stored back in the 'f' register. If the result is 1, the next instruction is executed. If the result is 0, the the next instruction is discarded and a NOP is executed instead, making this a $2T_{CY}$ instruction.

Commento ed Esempi

L'istruzione esegue due azioni: decremento del contenuto del registro indicato in argomento e successivo *skip* dell'istruzione seguente se il risultato del decremento vale 0. Tale istruzione viene utilizzata per costruire dei rudimentali cicli a test finale con indice a decremento. Un esempio è dato dal codice seguente:

Listing 5.33: Esempio 1 di DECFSZ

```
1.       movlw   5       ;Numero di iterazioni del ciclo
2.       movwf   Pippo   ;Contatore di ciclo
3. Qui:  ...             ;Corpo del ciclo
4.       ...
5.       decfsz  Pippo   ;Decremento del contatore di ciclo
6.       goto    Qui     ;Se Pippo > 0, torna a inizio ciclo
7.       ...             ;Se Pippo = 0, esce dal ciclo
```

Il corpo del ciclo viene eseguito, nel presente esempio, 5 volte. Alla quinta iterazione il contatore di ciclo (`Pippo`) diventa uguale a zero e il `goto Qui` viene saltato, uscendo dal ciclo.

La `decfsz` dura un ciclo macchina se il risultato del decremento è diverso da zero (dato che non è necessario eseguire alcuno *skip*), mentre dura due cicli macchina se il risultato del decremento vale zero.

Si noti un particolare curioso: mentre la `decf` modifica il *flag* di Zero la `decfsz` lo lascia inalterato.

5.1.16 GOTO

GOTO	Unconditional Branch
Syntax:	[*label*] GOTO k
Operands:	$0 \leq k \leq 2047$
Operation:	k $\rightarrow$ PC<10:0>, (PCLATH<4:3>) $\rightarrow$ PC<12:11>
Status Affected:	None
Description:	GOTO is an unconditional branch. The eleven-bit immediate address is loaded into PC bits <10:0>. The upper bits of the PC are loaded from PCLATH. GOTO is a two-cycle instruction.

Commento ed Esempi

L'istruzione esegue un salto, assoluto o relativo a seconda della sintassi, modificando quindi il PC, per cui l'istruzione dura sempre due cicli macchina.

I salti assoluti sono quelli effettuati verso un indirizzo definito esplicitamente, come nelle due seguenti istruzioni:

Listing 5.34: Esempio 1 di GOTO

```
        goto    0x01A3  ;Salta all'indirizzo 0x01A3
        ...
0x01A3: ...             ;Si notino i due punti
        goto    0x01A3  ;Torna all'indirizzo 0x01A3
        ...
        goto    Qui     ;Salta all'indirizzo indicato da Qui
        ...
        ...
 Qui:   ...             ;Si notino i due punti
        goto    Qui     ;Torna all'indirizzo indicato da Qui
        ...
```

Il primo caso è il più semplice, anche se il meno usato: l'indirizzo è espresso esplicitamente ed è compito del programmatore individuare correttamente l'indirizzo assoluto di destinazione ove saltare. Ciò non è sempre semplice e si sconsiglia tale pratica se non in casi eccezionali, in cui definire l'indirizzo di destinazione è particolarmente semplice ed immediato.

Il secondo caso prevede un utilizzo più semplice dell'istruzione, dato che l'indirizzo di destinazione è indicato mediante un'etichetta. Non è necessario, quindi, conoscere l'esatto indirizzo, giacché provvede l'assemblatore a calcolare l'indirizzo assoluto. Naturalmente l'etichetta di destinazione deve essere univoca.

Si noti che, in ambedue i casi, è possibile saltare in "avanti" verso indirizzi crescenti oppure "indietro" verso indirizzi decrescenti.

E' possibile esprimere il salto anche in maniera relativa, anche se si tratta più di un retaggio di programmazione *d'antan* che altro.

I salti relativi sono quelli effettuati rispetto alla posizione attuale. Si modifica il PC in modo da saltare in avanti o indietro di *n* posizioni *rispetto quella attuale*. Le seguenti istruzioni chiariranno il concetto:

Listing 5.35: Esempio 2 di GOTO

```
1.      ...           ;                                 (-1)
2.      goto    $-1   ;Torna all'indirizzo precedente   (+0)
3.      ...
4.      goto    $+3   ;Salta le prossime due istruzioni (+0)
5.      ...           ;                                 (+1)
6.      ...           ;                                 (+2)
7.      ...           ;Riprende da qui                  (+3)
```

La sintassi evidenziata all’istruzione 2. indica uno *spiazzamento* negativo di un’unità. Il PC viene modificato sottraendo 1 all’indirizzo attuale. L’istruzione 4. indica uno spiazzamento positivo di 3 unità, il cui conteggio avviene come indicato nell’esempio.

Si tratta, comunque, di una sintassi non molto utilizzata. Si noti che il compilatore traduce il salto relativo sempre in un salto assoluto, per cui non vi sono variazioni di efficienza del codice.

5.1.17 INCF

INCF	Increment f
Syntax:	[*label*] INCF f,d
Operands:	$0 \leq f \leq 127$ $d \in [0,1]$
Operation:	(f) + 1 $\rightarrow$ (destination)
Status Affected:	Z
Description:	The contents of register 'f' are incremented. If 'd' is 0, the result is stored in the W register. If 'd' is 1, the result is stored back in the 'f' register.

Commento ed Esempi

L'istruzione esegue l'incremento del *file register* indicato in argomento. Essa rappresenta l'istruzione duale della `decf` e quanto detto a proposito dei decrementi in notazione *unsigned* e in complemento a due rimane valido anche in questo caso. Si veda il seguente esempio, supponendo inizialmente `Pippo = B'11111110'`.

Listing 5.36: Esempio 1 di INCF

```
1.      incf    Pippo ;Dopo l'istruzione Pippo = B'11111111'
2.      incf    Pippo ;Dopo l'istruzione Pippo = B'00000000'
3.      incf    Pippo ;Dopo l'istruzione Pippo = B'00000001'
4.      incf    Pippo ;Dopo l'istruzione Pippo = B'00000010'
```

Anche in questo caso si ha il settaggio del *flag* di Zero dopo l'esecuzione dell'istruzione 3.

5.1.18 INCFSZ

INCFSZ	Increment f, Skip if 0
Syntax:	[*label*] INCFSZ f,d
Operands:	$0 \leq f \leq 127$ $d \in [0,1]$
Operation:	(f) + 1 $\rightarrow$ (destination), skip if result = 0
Status Affected:	None
Description:	The contents of register 'f' are incremented. If 'd' is 0, the result is stored in the W register. If 'd' is 1, the result is stored back in the 'f' register. If the result is 1, the next instruction is executed. If the result is 0, the the next instruction is discarded and a NOP is executed instead, making this a $2T_{CY}$ instruction.

Commento ed Esempi

L'istruzione esegue due azioni: incremento del contenuto del registro indicato in argomento e successivo *skip* dell'istruzione seguente se il risultato dell'incremento vale 0. Si tratta, quindi, dell'istruzione duale della `decfsz`. Essa viene utilizzata per costruire dei rudimentali cicli a test finale con indice ad incremento. Un esempio è dato dal codice seguente:

Listing 5.37: Esempio 1 di INCFSZ

```
1.       movlw   0xF0    ;Numero iterazioni del ciclo = 0xFF-W+1
2.       movwf   Pippo   ;Contatore di ciclo
3. Qui:  ...             ;Corpo del ciclo
4.       ...
5.       incfsz  Pippo   ;Incremento del contatore di ciclo
6.       goto    Qui     ;Se Pippo <> 0, torna a inizio ciclo
7.       ...             ;Se Pippo = 0, esce dal ciclo
```

Il corpo del ciclo viene eseguito, nel presente esempio, 16 volte. Alla sedicesima iterazione il contatore di ciclo (`Pippo`) diventa uguale a zero e il `goto Qui` viene saltato, uscendo dal ciclo.

La `incfsz` dura un ciclo macchina se il risultato dell'incremento è diverso da zero (dato che non è necessario eseguire alcuno *skip*), mentre dura due cicli macchina se il risultato dell'incremento vale zero.

Anche la `incfsz`, come la `decfsz` lascia inalterato il *flag* di Zero.

5.1.19 IORLW

IORLW	Inclusive OR Literal with W
Syntax:	[*label*] IORLW k
Operands:	$0 \leq k \leq 255$
Operation:	(W) .OR. k $\rightarrow$ (W)
Status Affected:	Z
Description:	The contents of the W register are OR'ed with the eight bit literal 'k' and the result is placed in the W register.

Commento ed Esempi

L'istruzione indica l'OR inclusivo, ovvero la disgiunzione inclusiva. Si supponga che prima dell'istruzione si abbia `W = 0xAA`.

Listing 5.38: Esempio 1 di IORLW

```
iorlw    0x55   ;Esegue l'OR logico fra W e 0x55
```

Siccome, dopo l'esecuzione dell'istruzione, si ha `W = 0xFF`, il *flag* di Zero vale `Z = 0`.

Se, invece, si suppone `W = 0x00`, l'istruzione

Listing 5.39: Esempio 2 di IORLW

```
iorlw    0x00   ;Esegue l'OR logico fra W e 0x00
```

dà come risultato `W = 0`, per cui `Z = 1`.

5.1.20 IORWF

IORWF	Inclusive OR W with f
Syntax:	[*label*] IORWF f,d
Operands:	$0 \leq f \leq 127$
	$d \in [0,1]$
Operation:	(W) .OR. (f) $\rightarrow$ (destination)
Status Affected:	Z
Description:	Inclusive OR the W register with register 'f'. If 'd' is 0, the result is stored in the W register. If 'd' is 1, the result is stored back in the 'f' register.

Commento ed Esempi

Si supponga `W = 0x00` e `Pippo = 0x55`.

Listing 5.40: Esempio 1 di IORWF

```
iorwf    Pippo  ;Esegue l'OR logico fra W e Pippo.
                ;Il risultato e' posto in Pippo.
```

Si noti che il risultato viene posto in `Pippo`, per cui `Pippo` non cambia e `Z = 0`.

Si supponga sempre `W = 0x00` e `Pippo = 0x55`.

Listing 5.41: Esempio 2 di IORWF

```
iorwf    Pippo,W;Esegue l'OR logico fra il contenuto
                ;W e Pippo. Il risultato e' posto in W.
```

In questo caso il risultato viene posto in `W`, per cui si ha che `W = 0x55` e `Pippo = 0x55`. Il *flag* di Zero non viene settato.

L'analisi fatta per l'istruzione `iorlw` è valida anche per la presente istruzione.

5.1.21 MOVF

MOVF	Move f
Syntax:	[*label*] MOVF f,d
Operands:	$0 \leq f \leq 127$
	$d \in [0,1]$
Operation:	(f) $\rightarrow$ (destination)
Status Affected:	Z
Description:	The contents of register 'f' are moved to a destination dependant upon the status of 'd'. If 'd' = 0, destination is W register. If 'd' = 1, the destination is file register itself. This condition is useful to test a file register, since status flag Z is affected.

Commento ed Esempi

L'istruzione permette di copiare[1] il contenuto del *file register* in W oppure in se stesso. Ovvero:

Listing 5.42: Esempio 1 di MOVF

```
1.      movf    Pippo,W ;Copia il contenuto di Pippo in W
2.      movf    Pippo   ;A cosa serve?
```

L'istruzione 1. permette, appunto, la copia del contenuto di `Pippo` in `W` con settaggio del *flag* di Zero se il contenuto di `W` vale zero.

L'istruzione 2. sembra apparentemente senza senso, dato che copia il contenuto di `Pippo` in `Pippo`. L'utilità sta nel fatto che se il contenuto del registro vale zero viene settato il relativo *flag*: è quindi utile a valutare velocemente se il contenuto di un registro vale zero.

[1]Il mnemonico *movf* sta per *move to file register*, quindi, tecnicamente, si dovrebbe usare il verbo "muovere", ovvero trasferire, piuttosto che "copiare". Si preferisce usare, però, il termine "copiare" per sottolineare che il valore non viene cancellato dal registro sorgente ma effettivamente copiato.

5.1.22 MOVLW

MOVLW	Move Literal to W
Syntax:	[*label*] MOVLW k
Operands:	$0 \leq k \leq 255$
Operation:	$k \rightarrow (W)$
Status Affected:	None
Description:	The eight bit literal 'k' is loaded into W register, The don't cares will assemble as 0's.

Commento ed Esempi

L'istruzione permette di caricare nel registro `W` un valore costante. Si veda il seguente codice:

Listing 5.43: Esempio 1 di MOVLW

```
movlw   0x55    ;Carica 0x55 in W
movlw   kPippo  ;Carica kPippo in W
movlw 0 ;Il flag di Zero non viene settato
```

La costante da caricare in W può essere espressa direttamente oppure in forma di etichetta. Si noti che si è anteposto all'etichetta la lettera `k` ad indicare che l'etichetta rappresenta una costante. Non si tratta di un obbligo, si badi bene, ma di una buona norma per una corretta ed ordinata programmazione.

L'ultima istruzione illustrata in esempio sottolinea come il *flag* di Zero non venga settato se si carica 0 in `W`.

5.1.23 MOVWF

MOVWF	Move W to f
Syntax:	[*label*] MOVWF f
Operands:	$0 \leq f \leq 127$
Operation:	$(W) \rightarrow (f)$
Status Affected:	None
Description:	Data is moved from W register to 'f' register.

Commento ed Esempi

L'istruzione permette di copiare il contenuto del registro `W` nel registro indicato in argomento. Si supponga che `W = 0x55`, allora:

Listing 5.44: Esempio 1 di MOVWF

```
movwf   Pippo   ;Copia W in Pippo (Pippo = 0x55)
clrw
movwf   Pippo   ;Il flag di Zero viene settato?
```

Si noti che la `movwf` lascia il *flag* di Zero invariato, qualsiasi sia il valore copiato nel registro destinazione.

5.1.24 NOP

NOP	No Operation
Syntax:	[*label*] NOP
Operands:	None
Operation:	No operation
Status Affected:	None
Description:	No operation

Commento ed Esempi

Sembra un'istruzione assolutamente inutile ed inutilizzata. Svolge, invece, un'azione molto importante: lascia passare un tempo noto, pari a un ciclo macchina, senza modificare alcunché nei registri del micro, a parte il PC, naturalmente. Supponendo un clock a 4MHz il sottostante codice introduce un ritardo di $3\mu s$:

Listing 5.45: Esempio 1 di NOP

```
nop             ;3x1 microsecondi di ritardo
nop
nop
```

5.1.25 RETFIE

RETFIE	Return from Interrupt
Syntax:	[*label*] RETFIE
Operands:	None
Operation:	TOS → PC, 1 → GIE
Status Affected:	None
Description:	Return from interrupt service routine. The stack is POPed and the Top Of the Stack is loaded into the program counter. The status GIE bit is set. This is a two-cycle instruction.

Commento ed Esempi

Permette il ritorno dopo una chiamata all'*interrupt service subroutine* (vedi 1.7.1 a pagina 18). Si veda il seguente codice

Listing 5.46: Esempio 1 di RETFIE

```
0x0123  movlw   0x55
0x0124  movwf   Pippo
0x0125  ...     ;Chiamata a ISR
        ...
        ...
ISR:    ;Interrupt Service Routine
        ;il registro W
        ...
        ...
        retfie  ;Ritorna all'istruzione seguente la
                ;interruzione
```

L'istruzione necessita di due cicli macchina per essere eseguita, dato che modifica il PC. Nel dettaglio avviene quanto segue.

Si supponga che dopo l'istruzione `movwf Pippo` il normale flusso dell'esecuzione del programma venga interrotto da una chiamata alla routine di servizio dell'interruzione. Prima di effettuare il salto all'indirizzo `0x0004` ove risiede il vettore di interruzione viene salvato l'indirizzo di ritorno, ossia `0x0125` nello *Stack*. Poi avviene il salto al vettore di interruzione e da lì all'indirizzo della *interrupt service routine*.

Detta routine viene eseguita e al termine di detto sottoprogramma viene eseguita la `retfie`. Essa, sostanzialmente, esegue due azioni: carica l'indirizzo di ritorno ove saltare, ripristinando lo stato dello *Stack* (viene cioè eseguito un *pop*) e setta il *flag* di abilitazione generale delle interruzioni (GIE) in modo da essere pronto ad accettare altre interruzioni.

Si noti che l'istruzione che era stata interrotta viene eseguita completamente prima di effettuare la procedura di salto all'ISR testé descritta.

5.1.26 RETLW

RETLW	Return with Literal in W
Syntax:	[*label*] RETLW k
Operands:	$0 \leq k \leq 255$
Operation:	$k \rightarrow (W)$, $TOS \rightarrow PC$
Status Affected:	None
Description:	The W register is loaded with the eight bit literal 'k'. The stack is POPed and the Top Of the Stack is loaded into the program counter. This is a two-cycle instruction.

Commento ed Esempi

Permette il ritorno da chiamata a subroutine, dopo aver caricato in `W` il valore della costante indicata in argomento. Si veda il seguente esempio.

Listing 5.47: Esempio 1 di RETLW

```
        movlw   0xAA
        movwf   Pippo   ;Pippo = 0xAA
        call    Sub     ;Chiamata a subroutine
        movwf   Pippo   ;Pippo = 0x55
        ...
        ...
Sub:    ;Subroutine
        ...
        ...
        retlw   0x55
```

Alla luce di quanto detto si può migliorare il codice presentato in 1.5 a pagina 16 scrivendo un semplice codice che calcola il quadrato del contenuto di `W`:

Listing 5.48: Esempio 2 di RETLW

```
        ...             ;Valore in W
        call    Sqr     ;Chiamata alla tabella
        ...             ;In W c'e' il risultato
        ...
Sqr:    addwf   PCL     ;Somma W al PC
        retlw   0       ;Tabella dei quadrati
        retlw   1
        retlw   4
        retlw   9
        retlw   16
        retlw   25
        ...
```

L'istruzione dura due cicli macchina.

5.1.27 RLF

RLF	Rotate Left f through Carry
Syntax:	[*label*] RLF f,d
Operands:	$0 \leq f \leq 127$ $d \in [0,1]$
Operation:	See description below
Status Affected:	C
Description:	The contents of register ‘f’ are rotated one bit to the left through the Carry Flag. If ‘d’ is 0, the result is stored in the W register. If ‘d’ is 1, the result is stored back in the ‘f’ register.

Commento ed Esempi

L’istruzione esegue la rotazione verso sinistra, attraverso il *flag* di Carry, del contenuto del registro indicato in argomento. L’azione può essere graficamente illustrata come in fig. 5.1.

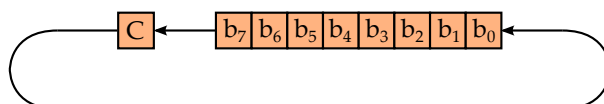



Figura 5.1: Rotate Left through Carry

I singoli bit del *file register* indicato in argomento vengono spostati verso sinistra. Ciò significa che il bit 7 viene spostato nel Carry, il bit 6 viene spostato nel bit bit 7, il bit 5 nel bit 6 e così via. Il valore originale che il Carry aveva prima dello spostamento viene posto nel bit 0, dando compimento ad una “rotazione” antioraria (a sinistra) dei bit attraverso il *flag* di Carry.

Naturalmente il *flag* di Carry viene modificato, ma non il *flag* di Zero, anche se Carry e contenuto del registro dovessero essere entrambi a 0 prima della rotazione.

Il risultato della rotazione può essere memorizzato sia nel registro che in `W`, come evidenziato nelle seguenti istruzioni. Si supponga inizialmente `C = 1` e `Pippo = 0x55`. Il risultato delle operazioni è indicato nei commenti alle istruzioni[2].

Listing 5.49: Esempio 1 di RLF

```
rlf     Pippo   ;Pippo = 0xAB e C = 0
rlf     Pippo,W ;Pippo = 0xAB, C = 1 e W = 0x56
```

[2]Si noti che trattandosi di pagine didattiche l’autore insiste nel commentare quali siano gli effetti dell’istruzione, ossia *cosa* fa. Sappia lo studente, però, che i commenti di un programma non tendono a rispondere a *cosa* fa un’istruzione (è evidente che il programmatore lo sa), bensì *perché* si è scelta una tal soluzione.

5.1.28 RETURN

RETURN	Return from Subroutine
Syntax:	[*label*] RETURN
Operands:	None
Operation:	TOS $\rightarrow$ PC
Status Affected:	None
Description:	Return from subroutine. The stack is POPed and the Top Of the Stack is loaded into the program counter. This is a two-cycle instruction.

Commento ed Esempi

Permette il ritorno da chiamata a subroutine. Si veda il seguente esempio.

Listing 5.50: Esempio 1 di RETURN

```
        movlw   0xAA
        movwf   Pippo   ;Pippo = 0xAA
        call    Sub     ;Chiamata a subroutine
        movwf   Pippo   ;Pippo = 0x55
        ...
        ...
Sub:    ;Subroutine
        ...
        movlw   0x55
        return          ;Ritorna all'istruzione seguente la
                        ;chiamata a subroutine
```

Modificando il PC, l'istruzione dura due cicli macchina.

5.1.29 RRF

RRF	Rotate Right f through Carry
Syntax:	[*label*] RRF f,d
Operands:	$0 \leq f \leq 127$ $d \in [0,1]$
Operation:	See description below
Status Affected:	C
Description:	The contents of register 'f' are rotated one bit to the right through the Carry Flag. If 'd' is 0, the result is stored in the W register. If 'd' is 1, the result is stored back in the 'f' register.

Commento ed Esempi

L'istruzione esegue la rotazione verso destra, attraverso il *flag* di Carry, del contenuto del registro indicato in argomento. L'azione può essere graficamente illustrata come in fig. 5.2.

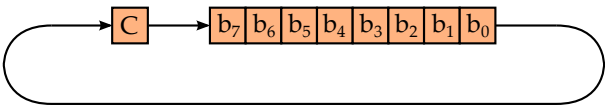



Figura 5.2: Rotate Right through Carry

I singoli bit del *file register* indicato in argomento vengono spostati verso destra. Ciò significa che il bit 1 viene spostato nel bit 0, il bit 2 viene spostato nel bit bit 1, il bit 3 nel bit 2 e così via. Il valore originale che il bit 0 aveva prima dello spostamento viene posto nel Carry, dando compimento ad una "rotazione" oraria (a destra) dei bit attraverso il *flag* di Carry.

Naturalmente il *flag* di Carry viene modificato, ma non il *flag* di Zero, anche se Carry e contenuto del registro dovessero essere entrambi a 0 prima della rotazione.

Il risultato della rotazione può essere memorizzato sia nel registro che in `W`, come evidenziato nelle seguenti istruzioni. Si supponga inizialmente `C = 1` e `Pippo = 0xAA`. Il risultato delle operazioni è indicato nei commenti alle istruzioni.

Listing 5.51: Esempio 1 di RRF

```
rrf     Pippo   ;Pippo = 0xD5 e C = 0
rrf     Pippo,W ;Pippo = 0xD5, C = 1 e W = 0x6A
```

5.1.30 SLEEP

SLEEP	Sleep
Syntax:	[*label*] SLEEP
Operands:	None
Operation:	00h $\rightarrow$ WDT, 0 $\rightarrow$ WDT prescaler, 1 $\rightarrow \overline{TO}$, 0 $\rightarrow \overline{PD}$
Status Affected:	$\overline{TO}$, $\overline{PD}$
Description:	The power-down status bit, $\overline{PD}$ is cleared. Time-out status bit, $\overline{TO}$ is set. Watchdog Timer and its prescaler are cleared. The processor is put into SLEEP mode with the oscillator stopped.

Commento ed Esempi

L'istruzione permette di porre il micro in modalità di basso consumo, permettendo un assorbimento minore di $1\mu A$. In tale modalità il micro interrompe ogni attività, rimanendo in attesa di essere risvegliato da uno dei seguenti eventi:

- Reset esterno attraverso il pin $\overline{MCLR}$;
- *Wake up* per fine conteggio del WDT, se abilitato;
- Interruzione lanciata dal pin INT, dal PORTB o da determinate periferiche. L'argomento verrà affrontato dettagliatamente nella sezione **??**.

Listing 5.52: Esempio 1 di SLEEP

```
sleep           ;Il micro si pone in stand by
...             ;In seguito ad un wake up il micro
                ;riprende dall'istruzione seguente che
                ;lo ha posto in stand by
```

Il bit $\overline{PD}$ del registro di stato è posto a 0, mentre il bit $\overline{TO}$ è posto a 1. Il WDT e l'eventuale prescaler ad esso abbinato sono azzerati.

Le informazioni fornite nella presente pagina sono assolutamente sommarie, dato che l'argomento è piuttosto complesso. Esso verrà trattato esaustivamente nella sezione **??**.

5.1.31 SUBLW

SUBLW	Subtract W from Literal
Syntax:	[*label*] SUBLW k
Operands:	$0 \leq k \leq 255$
Operation:	k - (W) → (W)
Status Affected:	C, DC, Z
Description:	The contents of the W register are subtracted (2's complement method) from the eight bit literal 'k' and the result is placed in the W register.

Commento ed Esempi

Si supponga che prima dell'istruzione si abbia `W = 7`.

Listing 5.53: Esempio 1 di SUBLW

```
sublw   9       ;Sottrae da 9 il contenuto di W
```

Al termine dell'istruzione si ha `W = 2`. Si noti che la sottrazione viene tecnicamente eseguita mediante somma in complemento a due, per cui il contenuto di `W` viene complementato e sommato al valore costante. La cifra più significativa, ossia il nono bit, viene trascurato ed i restanti otto bit posti nel registro `W`.

Il prossimo esempio continua l'analisi delle operazioni in complemento a due.

Listing 5.54: Esempio 2 di SUBLW

```
movlw   0xFE
sublw   1       ;Quanto vale W dopo la sottrazione?
```

Siccome l'operazione è effettuata in complemento a due *entrambi* gli operandi sono considerati in complemento a due. Ciò significa che il contenuto di `W` prima della sottrazione è un numero negativo (-2) e l'operazione diventa

$$0x01 - 0xFE \quad \rightarrow \quad 1 - (-2) = 3 \tag{5.1}$$

Il prossimo esempio esamina il caso in cui entrambi numeri siano negativi:

Listing 5.55: Esempio 3 di SUBLW

```
movlw   0xFE
sublw   0xFD    ;W = 0xFF dopo la sottrazione, ossia -1
```

Infatti

$$0xFD - 0xFE \quad \rightarrow \quad (-3) - (-2) = -1 \tag{5.2}$$

Un esempio di *overflow* è dato dal prossimo codice:

Listing 5.56: Esempio 4 di SUBLW

```
movlw   0x02
sublw   0x81    ;W = 0x7F dopo la sottrazione, ossia +127
```

Sottraendo un numero positivo da un numero negativo si deve ottenere ancora un numero negativo, di modulo maggiore rispetto al minuendo, ed invece il risultato risulta essere positivo (+127). Quindi il risultato è palesemente errato. Alle stesse conclusioni si giunge analizzando i riporti del Segno e del Carry, che sono discordi e rimarcano l'*overflow*.

Piuttosto curiosa è l'ultima operazione:

Listing 5.57: Esempio 4 di SUBLW

```
movlw   0x81
sublw   0x02    ;Quanto vale W dopo la sottrazione?
```

L'operazione, convertita in decimale, diventa

$$0x02 - 0x81 \quad \rightarrow \quad 2 - (-127) = 129 \quad \rightarrow \quad 0x81 \tag{5.3}$$

Sembra cioè che la sottrazione lasci immutato il registro W, che valeva 0x81 prima della sottrazione e vale ancora 0x81 dopo la sottrazione (?). Ciò è dovuto al supero di capacità prodotto dalla sottrazione, con conseguente perdita di significato del risultato.

Si inciampa facilmente nelle operazioni in complemento a due, per cui si rinnova il consiglio di un attento studio dell'argomento.

5.1.32 SUBWF

SUBWF	Subtract W from f
Syntax:	[*label*] SUBWF f,d
Operands:	$0 \leq f \leq 127$ $d \in [0,1]$
Operation:	(f) - (W) $\rightarrow$ (destination)
Status Affected:	C, DC, Z
Description:	The contents of the W register are subtracted (2's complement method) from the 'f' register. If 'd' is 0, the result is stored in the W register. If 'd' is 1, the result is stored back in the 'f' register.

Commento ed Esempi

Si supponga che prima dell'istruzione si abbia `W = 7` e che `Pippo = 9`.

Listing 5.58: Esempio 1 di SUBWF

```
subwf    Pippo  ;Sottrae W dal contenuto di Pippo
```

Il risultato viene posto in `Pippo`: `Pippo = 2`.

Si supponga sempre `W = 7` e `Pippo = 9`.

Listing 5.59: Esempio 2 di SUBWF

```
subwf    Pippo,W;Ora il risultato e' posto in W
```

In questo caso si ottiene `Pippo = 9` e `W = 2`.

Quanto detto per l'istruzione `sublw` rimane valido anche per la presente istruzione.

5.1.33 SWAPF

SWAPF	Swap Nibbles in f
Syntax:	[*label*] SWAPF f,d
Operands:	$0 \leq f \leq 127$ $d \in [0,1]$
Operation:	(f<3:0>) $\rightarrow$ (destination<7:4>) (f<7:4>) $\rightarrow$ (destination<3:0>)
Status Affected:	None
Description:	The upper and lower nibbles of register 'f' are exchanged. If 'd' is 0, the result is stored in the W register. If 'd' is 1, the result is stored back in the 'f' register.

Commento ed Esempi

L'istruzione permette lo scambio dei due *nibble*[3] del registro indicato in argomento, come evidenziato nell'esempio seguente. Si supponga inizialmente `Pippo = 0x5A`:

Listing 5.60: Esempio 1 di SWAPF

```
        swapf    Pippo  ;Dopo l'istruzione si ha Pippo = 0xA5
```

L'azione può essere resa graficamente com in fig. 5.3. Si noti come la posizione relativa dei singoli bit all'interno dei due *nibble* non cambi.

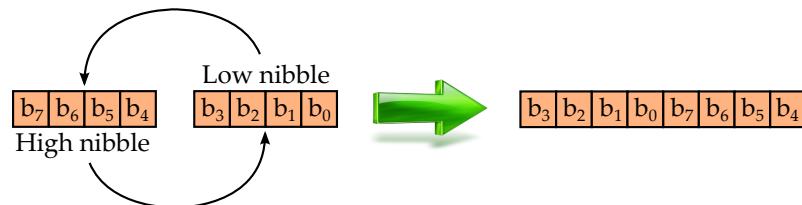



Figura 5.3: Swap File Register

La particolarità di quest'istruzione è data dal fatto che non altera i *flag* di stato, per cui torna utile nel salvataggio dei registri fondamentali durante le interruzioni. L'argomento sarà discusso nella sezione **??**.

[3]Si ricorda che il *nibble* è formato da 4 bit. Due *nibble* formano un byte. Il *low nibble* è formato dai 4 bit meno significativi, mentre il *high nibble* è formato dai 4 bit più significativi del byte.

5.1.34 XORLW

XORLW	Exclusive OR Literal with W
Syntax:	[*label*] XORLW k
Operands:	$0 \leq k \leq 255$
Operation:	(W) .XOR. k $\rightarrow$ (W)
Status Affected:	Z
Description:	The contents of the W register are XOR'ed with the eight bit literal 'k' and the result is placed in the W register.

Commento ed Esempi

L'istruzione esegue la XOR, ovvero la disgiunzione esclusiva, fra il contenuto del registro `W` e la costante. Si supponga che prima dell'istruzione si abbia `W = 0xFF`.

Listing 5.61: Esempio 1 di XORLW

```
xorlw    0x55   ;Esegue la XOR logica fra W e 0x55
```

Siccome, dopo l'esecuzione dell'istruzione, si ha `W = 0xAA`, il *flag* di Zero vale `Z = 0`. Si noti come la XOR fra un valore e 0xFF equivalga alla negazione (complemento a uno) del valore stesso.

Se, invece, si suppone `W = 0x55`, l'istruzione

Listing 5.62: Esempio 2 di XORLW

```
xorlw    0x55   ;W e la costante sono uguali
```

dà come risultato `W = 0`, per cui `Z = 1`.

5.1.35 XORWF

XORWF	Exclusive OR W with f
Syntax:	[*label*] XORWF f,d
Operands:	$0 \leq f \leq 127$ $d \in [0,1]$
Operation:	(W) .XOR. (f) $\rightarrow$ (destination)
Status Affected:	Z
Description:	Exclusive OR the W register with register 'f'. If 'd' is 0, the result is stored in the W register. If 'd' is 1, the result is stored back in the 'f' register.

Commento ed Esempi

Si supponga `W = 0xAA` e `Pippo = 0x55`.

Listing 5.63: Esempio 1 di XORWF

```
xorwf    Pippo  ;Il risultato e' posto in Pippo.
```

Si avrà `Pippo = 0xFF`, per cui `Z = 0`.

Si supponga nuovamente `W = 0xAA` e `Pippo = 0x55`.

Listing 5.64: Esempio 2 di XORWF

```
xorwf    Pippo,W;Il risultato e' posto in W.
```

In questo caso il risultato viene posto in `W`, per cui si ha che `W = 0xFF` e `Pippo = 0x55`. Il *flag* di Zero non viene settato.

5.2 La scelta delle etichette

Sicuramente lo studente avrà notato, magari sorridendo maliziosamente, la particolare scelta delle etichette fatta nelle sezioni dedicate alle istruzioni: imperano incontrastate le etichette `Pippo` e `Qui`.

La scelta non è casuale né vezzosa.

Le etichette possono rappresentare una costante, una variabile o un'espressione utilizzate nel programma. Possono anche indicare un indirizzo di istruzione.

Nel sottostante codice, `Pippo`, `kPippo` e `Qui` sono tutte etichette valide e rappresentano rispettivamente una variabile, una costante ed un indirizzo di istruzione.

Listing 5.65: Esempio di etichette

```
        movlw   kPippo  ;kPippo: costante
        movwf   Pippo   ;Pippo: variabile
Qui:    nop             ;Qui: indirizzo istruzione
        goto    Qui     ;Qui: indirizzo istruzione
```

Le etichette devono osservare delle regole sintattiche e *dovrebbero* osservare anche delle regole mnemoniche.

5.2.1 Le regole sintattiche

La Microchip ha adottato, per la definizione delle etichette in assembly, le regole sintattiche del linguaggio C, con qualche piccola variante. La scrittura di dette etichette deve soddisfare le seguenti regole:

- i caratteri uitilizzabili possono essere solamente le lettere maiuscole e minuscole dell'alfabeto inglese (A..Z, a..z), le cifre (0..9), l'*underscore* (_) e il punto di domanda (?);
- un'etichetta non deve mai iniziare con una cifra;
- non si possono usare le parole chiave come etichetta.

Sono quindi etichette valide/non valide le seguenti:

Valide			Non valide		
Pippo	Pippo_1	Pippo?	3M	nop	0hhh

La prima delle etichette non valide (3M) inizia con cifra, per cui genera un errore di tipo 108 (*Illegal character*). La seconda etichetta non è valida perché `nop` è una parola chiave (in questo caso un'istruzione mnemonica), mentre la terza etichetta inizia con cifra, anche se lo zero può sembrare una O maiuscola.

5.2.2 Le regole mnemoniche

Le regole mnemoniche sono più difficili da enunciare. Innanzi tutto non si tratta di regole vere e proprie, ma semplicemente di buon senso. La Microsoft ha cercato di definire anche le regole mnemoniche per il proprio ambiente di sviluppo Visual Studio, senza però renderle obbligatorie.

Prima di formalizzare alcuni consigli, si richiama l'attenzione sul seguente codice:

Listing 5.66: Subroutine PerDieci

```
Wx10:   ;Wx10 moltiplica il contenuto di W per 10 e pone il
        ;risultato nuovamente in W. Per ottenete cio' memorizza
        ;momentaneamente il valore di W in Temp e lo moltiplica
        ;per 4. Somma poi il valore di Temp in modo da ottenere
        ;W*5 e moltiplica il tutto per 2.
        movwf   Temp    ;Salva W
        movwf   Parziale

        ;Moltiplica Parziale per 4
        bcf     STATUS,C
        rrf     Parziale
        rrf     Parziale,W

        ;Somma il contenuto originale di W salvato in Temp
        addwf   Temp,W
        movwf   Parziale

        ;Moltiplica tutto per 2, ottenendo W*10
        rrf     Parziale,W
        return
```

L'etichetta della subroutine `Wx10` illustra sinteticamente cosa la routine fa, ossia moltiplicare per dieci il contenuto di `W`. Detto valore viene memorizzato in una variabile temporanea (`Temp`) e nel risultato parziale `Parziale`). I commenti illustrano poi i vari passi. Il codice così scritto risulta abbastanza leggibile e quando si troverà la subroutine `Wx10` si capirà immediatamente che essa serve a moltiplicare per dieci il contenuto di `W`.

Ben diverso effetto fa il seguente codice:

Listing 5.67: Subroutine Cut

```
Cut:    movwf   Raggio
        movwf   Lavandino
        bcf     STATUS,0
        rrf     Lavandino
        rrf     Lavandino,0
        addwf   Raggio,0
        movwf   Lavandino
        rrf     Lavandino,0
        return
```

Nemmeno il mago Merlino potrebbe capirci qualcosa da un codice simile. Prima di formulare regole cerchiamo di capire quali sono le parti di codice (o le mancanze) che lo rendono di difficile comprensione.

Innanzi tutto l'etichetta che dà il nome alla subroutine non illustra cosa essa fa. Quando l'ignaro lettore del programma legge l'istruzione `call Cut` probabilmente penserà a una qualche azione che ricordi il taglio o una divisione, oppure a qualche località indiana, ma certamente non ad un codice che moltiplica per dieci il contenuto di `W`. Questo è un primo errore che ostacola una veloce ed efficiente lettura/comprensione di un programma in assembly.

Il secondo errore che balza agli occhi di un programmatore esperto è una mancanza: la assoluta e totale mancanza di commenti che illustri il codice. Si tratta di un errore diffusissimo fra gli studenti, che sono spesso completamente concentrati sulle istruzioni, da dimenticare che ciò che scrivono potrà, un domani, essere letto da qualcuno.

I commenti sono assolutamente fondamentali per la comprensione di un codice: devono essere sintetici ed esaustivi e rispondere alla domanda **Perché**, non **Cosa**. Il seguente è un cattivo esempio di commento:

Listing 5.68: Commento inutile

```
movlw   100     ;Carico 100 in W. Lo spiego perche'
                ;il mio professore non e' in grado
                ;di capirlo da solo.
```

Il commento, in questo caso, spiega **cosa fa** l'istruzione. In certi casi può anche essere utile: quando, ad esempio, un insegnante spiega l'*instruction set* di un microcontrollore agli allievi. Diventa inutile (o addirittura offensivo nei confronti di chi legge) negli altri casi.

Un esempio di commento corretto è il seguente:

Listing 5.69: Commento utile

```
movlw   100     ;W contiene il numero di millisecondi
                ;di ritardo che la subroutine Ms deve
                ;eseguire. La routine verra' chiamata
                ;fra breve.
```

In questo caso il commento spiega **perché** si carica 100 in `W` e non, ad esempio, 101. Evidentemente la routine `Ms` esegue un ritardo espresso in millisecondi pari al valore contenuto in `W`.

Inoltre, il commento svolge anche un altro importante compito. Scrivendo il commento lo studente deve illustrare l'algoritmo che ha elaborato. Tanto più chiaramente avrà elaborato l'algoritmo e tanto meglio e più dettagliatamente esso verrà illustrato. Se il commento sarà simile all'aria fritta, lo studente non avrà definito con sufficiente dettaglio l'algoritmo e potrà facilmente rendersene conto rileggendo criticamente il commento.

Un terzo errore grave, per fortuna non diffusissimo fra gli studenti, è dato dalla scelta dei nomi delle variabili: `Raggio` e `Lavandino` non illustrano quale sia il loro compito all'interno del codice e quindi non ne facilitano la lettura.

I nomi delle variabili devono essere contestualizzati al compito che le variabili stesse sono chiamate a svolgere nel programma e illustrare brevemente il loro ruolo all'interno del codice.

Ecco perché, a volte, è utile definire le variabili con nomi "strani" come ad esempio `Pippo`. Chi legge capisce che il nome assegnato è privo di significato e volutamente decontestualizzato.

Lo studente scelga quindi con cura il nome delle etichette: il programma risulterà più semplice da leggere e sarà più facile trovare i bachi in fase di *debug*.

Si può, quindi, tentare di "regolamentare" la definizione mnemonica delle etichette:

1. si cerchi di contestualizzare il nome di una variabile o costante. Un contatore di ciclo può chiamarsi `Cnt`, `Counter` o `Cont`, ma possibilmente non `X` o `C` (perché no?);
2. le costanti dovrebbero iniziare con la lettera k (antico retaggio del Fortran);
3. quando le etichette sono composte (formate da più parole) si usano due tecniche distinte:
   (a) si separano mediante *underscore*, come ad esempio `Loop_cnt`;
   (b) si separano mediante maiuscole, come ad esempio `LoopCnt`.
4. è meglio evitare etichette troppo lunghe e difficili da leggere: si preferisce troncarle. Meglio scrivere `DelCnt` piuttosto che `Delay_counter` o, peggio ancora, `Contatore_di_ritardo`. L'assembly è *line oriented* e ciò consiglia etichette corte;
5. le subroutine svolgono "azioni" ed è buona norma utilizzare in tal caso, se possibile e mnemonico, etichette che derivino da verbi. Una routine di moltiplicazione potrebbe chiamarsi `Multiply` e una routine di ordinamento potrebbe chiamarsi `Sort` oppure `Ordina`;
6. le etichette degli indirizzi dovrebbero fornire una qualche indicazione sul tipo di azione che verrà fatta, come negli esempi seguenti:

Listing 5.70: Struttura selezione

```
                btfsc   Numero,0
                goto    SeDispari
                goto    SePari
                ...
                ...
SePari:         ...
                ...
SeDispari:      ...
```

Le suddette regole non sono regole sintattiche ma di "buon senso". Lo studente cerchi di assimilarle: i programmi risulteranno più comprensibili e conterranno meno errori.

5.3 Le direttive

Molto spesso lo studente fa confusione fra *direttive* e *istruzioni*. L'argomento è stato trattato in maniera sufficientemente dettagliata in altra documentazione ("Linguaggio C" dello stesso autore), per cui si rimanda a tale scritto per una trattazione più estesa. Nella presente sezione ci si limiterà ad introdurre il concetto, senza scendere nei dettagli.

Per poter definire il concetto di direttiva è prima necessario conoscere quello di preprocessore.

Definizione 1 (Preprocessore)**.**
Il preprocessore è un software applicativo che esegue delle operazioni sul testo di un codice sorgente. Tale applicazione viene eseguita prima della compilazione del codice sorgente.

Siccome il preprocessore "esegue delle operazioni", è necessario indicare l'operazione da eseguire:

Definizione 2 (Direttiva)**.**
La direttiva definisce l'operazione che il preprocessore deve eseguire sul codice sorgente.

Ora è possibile distinguere l'azione svolta dalla direttiva da quella dell'istruzione: la direttiva è eseguita dal preprocessore *prima* che il microprocessore esegua il programma scritto dall'utente; l'istruzione è eseguita dal microprocessore *durante* l'esecuzione del programma.

Nella presente sezione nonsi vogliono analizzare tutte le possibili direttive del linguaggio assembly, ma solamente due in particolare, che normalmente lo studente interpreta come istruzioni (ed in questo particolare caso sarebbe parzialmente scusato): la `banksel` e la `ibanksel`. Esse sono illustrare di seguito come se fossero delle istruzione, ma lo studente è invitato a ricordare la loro vera natura.[4]

[4]Vi è un aspetto che le presenti pagine non potranno chiarire: il numero di cili macchina che dette direttive implicano. Teoricamente, la **banksel** dobrebbe essere eseguita in due cicli macchina, mentre la **ibanksel** dovrebbe essere eseguita in un solo ciclo macchina, ma l'autore non è riuscito a trovare conferma di dette supposizioni nella pur poderosa documentazione Microchip.

5.3.1 banksel

banksel	Bank selection wich f belongs to
Syntax:	banksel f
Operands:	$0 \leq f \leq 511$
Operation:	(W) .XOR. (f) $\rightarrow$ (destination)
Status Affected:	None
Description:	Exclusive OR the W register with register 'f'. If 'd' is 0, the result is stored in the W register. If 'd' is 1, the result is stored back in the 'f' register.

5.3.2 bankisel

XORWF	Exclusive OR W with f
Syntax:	bankisel f
Operands:	$0 \leq f \leq 127$ $d \in [0,1]$
Operation:	(W) .XOR. (f) $\rightarrow$ (destination)
Status Affected:	Z
Description:	Exclusive OR the W register with register 'f'. If 'd' is 0, the result is stored in the W register. If 'd' is 1, the result is stored back in the 'f' register.

5.4 Esercizi

Gli esercizi riportati nelle seguenti pagine sono tutti relativi a quanto esposto nel capitolo 5. Le soluzioni degli esercizi sono riportate in Appendice A.

L'assembly del PIC16F877

1. ◊◊◊ Quante sono le istruzioni dell'assembly del PIC16F877?
2. ◊◊◊ Si elenchino quante più istruzioni possibili.
3. ◊◊◊ Si traducano le istruzioni in italiano.
4. ◊◊◊ Si elenchino le istruzioni aritmetiche.
5. ◊◊◊ Si elenchino le istruzioni logiche.
6. ◊◊◊ Si elenchino le istruzioni legate alle subroutine.
7. ◊◊◊ Si elenchino le istruzioni di *skip* e salto.
8. ◊◊◊ Si elenchino le istruzioni di spostamento (*move*).
9. ◊◊◊ Si spieghi la differenza fra una `movf` e una `movwf`.
10. ◊◊◊ Si decriva nei dettagli l'azione della `crlwdt`.
11. ◊◊◊ Si decriva nei dettagli l'azione della `comf`.
12. ◊◊◊ Si decriva nei dettagli l'azione della `retfie`.
13. ◊◊◊ Si decriva nei dettagli l'azione della `rrf`.
14. ◊◊◊ Si decriva nei dettagli l'azione della `swapf`.
15. ◊◊◊ La `sublw` esegue la differenza `(W)-k` o `k-(W)`? Si definisca una regola mnemonica per ricordarlo facilmente.
16. ◊◊◊ La `subwf` esegue la differenza `(W)-(f)` o `(f)-(W)`? Si definisca una regola mnemonica per ricordarlo facilmente.
17. ◊◊♦ Quali sono i *flag* modificati dalle istruzioni aritmetiche?
18. ◊◊♦ Quali sono i *flag* modificati dalle istruzioni logiche?
19. ◊◊♦ Quali sono i *flag* modificati dalle istruzioni di incremento/decremento?
20. ◊◊♦ Quali sono i *flag* modificati dalla `sleep`?
21. ◊◊◊ Quali sono le regole sintattiche per la definizione delle etichette?
22. ◊◊◊ Quali sono le caratteristiche mnemoniche che le etichette dovrebbero avere?
23. ◊◊◊ Quali sono le caratteristiche di un buon commento?

Appendice A

Il codice ASCII

ASCII è un acronimo che significa *American Standard Code for Information Interchange*. La sottostante tabella ASCII contiene i caratteri non visibili (dal primo al 31*esimo*) che hanno funzioni di controllo del testo (o della trama) ed i cosidetti caratteri tipografici dal 32*esimo* carattere in poi.

La colonna più a sinistra della tabella indica i 4 bit meno significativi del carattere, mentre la prima riga indica i 3 bit più significativi del carattere.

LSB/MSB	000	001	010	011	100	101	110	111
0000	NUL	DLE	SP	0	@	P	′	p
0001	SOH	DC1	!	1	A	Q	a	q
0010	STX	DC2	″	2	B	R	b	r
0011	ETX	DC3	#	3	C	S	c	s
0100	EOT	DC4	$	4	D	T	d	t
0101	ENQ	NAK	%	5	E	U	e	u
0110	ACK	SYN	&	6	F	V	f	v
0111	BEL	ETB	′	7	G	W	g	w
1000	BS	CAN	(	8	H	X	h	x
1001	HT	EM	)	9	I	Y	i	y
1010	LT	SUB	*	:	J	Z	j	z
1011	VT	ESC	+	;	K	[	k	{
1100	FF	FS	,	<	L	\	l	|
1101	CR	GS	-	=	M	]	m	}
1110	SO	RS	.	>	N	∧	n	∼
1111	SI	US	/	?	O	_	o	DEL

Tabella ASCII (7 bit code)

Bibliografia

[1] Knuth Donald Ervin, *The Art of Computer Programming*, Addison Wesley, Third Edition 1997

Articoli e *Application Notes*

[2] Palacherla Amar, Fink Scott, *AN555 - Software Implementation of Asynchronous Serial I/O*, Microchip Technology Inc., 1997

[3] Testa Frank, *AN575 - IEEE 754 Compliant Floating Point Routines*, Microchip Technology Inc., 1997

[4] Palmer Mark, *AN607 - Power-up Trouble Shooting*, Microchip Technology Inc., 1997

Sitografia

[5] AA.VV., *DS30292C - Datasheet PIC16F87x*, Microchip Technology Inc., 2001 consultabile al sito http://www.microchip.com